Clube Delphi

DAO aplicado
Explore as melhores práticas

FireDAC com DataSnap
Substitua o dbExpress
em aplicativos multicamadas

Edição 159 :: R$ 14,90

DEVMEDIA

DO DELPHI AO GOOGLE PLAY

Passo a passo para publicar seu APP

Mini curso: EntityDAC
Uma alternativa ao
TMS Aurelius

FireDAC em Cliente/Servidor
Reutilização de DataModule
VCL para FireMonkey

ISSN 1517990-7

DEVMEDIA

DÊ UM SALTO EM CONHECIMENTO!

Acesse o maior portal para desenvolvedores da América Latina!

20 mil posts

430 mil cadastrados

10 milhões de page-views por mês

http://www.devmedia.com.br/

DEVMEDIA

Clube Delphi

159ª Edição 2014 ISSN 1517990-7 Impresso no Brasil

Editor Geral

Paulo Quicoli (pauloquicoli@gmail.com)

Equipe Editorial

Guinther Pauli (guintherpauli@gmail.com)

Fabrício Hissao Kawata (fabricio.kawata@bol.com.br)

Giuliano Scombatti Pinto (giuliano@sygnux.com.br)

Daniel Sobrinho Laporte (daniel.laporte@gmail.com)

Jornalista Responsável

Kaline Dolabella - JP24185

Consultor Técnico

Davi Costa (davigc_08@hotmail.com)

Capa

Romulo Araujo

Diagramação

Janete Feitosa

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

www.devmedia.com.br/central
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

publicidade@devmedia.com.br

Distribuição

FC Comercial e Distribuidora S.A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Paulo Quicoli - Editor da Revista
pauloquicoli@gmail.com

Paulo Quicoli - **Editor Geral**

pauloquicoli@gmail.com
quicoli.wordpress.com
twitter.com/pauloquicoli

MVP

Assine agora e tenha acesso a todo o conteúdo da DevMedia: **www.devmedia.com.br/mvp**

DEVMEDIA

Sumário

Artigo no estilo Solução Completa

04 – Google Play Store: Criando, configurando e disponibilizando seu APP
[*Rafael Mota*]

Artigo no estilo Solução Completa

14 – DataSnap: Migrando do dbExpress para FireDAC
[*Fabrício Hissao Kawata*]

Artigo no estilo Curso, Conteúdo sobre Arquitetura

24 – EntityDAC: Ferramenta de Mapeamento Objeto-Relacional – Parte Final
[*Fabrício Hissao Kawata*]

Conteúdo sobre Boas Práticas

32 – DAO Pattern: Persistência de Dados utilizando o padrão DA
[*Filipe Dalepiane*]

Artigo no estilo Solução Completa

42 – FireDAC CRUD: Cadastro completo utilizando VCL e FireMonkey
[*Filipe Dalepiane*]

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Google Play Store: aprenda a criar, configurar e disponibilizar seu APP

A Google Play – conhecida anteriormente por Android Market – é a loja online da gigantesca e conhecida Google, empresa que lidera a Open Handset Alliance, um grupo formado por nomes de peso da tecnologia móvel, operadoras telefônicas, empresas de software e fabricantes de smartphones. A OHA, como também é conhecida esta aliança, é a criadora, maior investidora, incentivadora e divulgadora da plataforma Android no mundo.

Estima-se que já ultrapasse mais de um milhão de aplicativos disponíveis na loja para download e que o número de downloads já tenha superado a incrível marca de cinquenta bilhões, contabilizando os gratuitos e pagos, movimentando um mercado bilionário. Sabendo disso, fica mais do que claro que nossos aplicativos devem estar nessa vitrine mundial, seja para venda ou apenas para portfólio. Mas para chegar até lá, antes devemos executar uma série de tarefas e configurações no nosso projeto Delphi e no Developer Console. Os mais importantes e obrigatórios serão abordados neste artigo.

O objetivo aqui será criar um aplicativo para armazenar contatos telefônicos em um banco de dados SQLite, banco este que iremos criar usando o próprio IDE do Delphi, sem a necessidade de um gerenciador de banco de dados. Será abordado, por exemplo, o que é e como gerar o Keystore, veremos algumas regras que a Google exige que seu App cumpra para que possa ser publicado; conheceremos pequenas dicas que poderão tornar seu aplicativo mais visível ao público, aumentando a chance de downloads, e por fim, conheceremos o Google Play Developer Console – ferramenta usada para publicar e gerenciar todos os seus aplicativos. Ao terminar esta breve leitura, você terá conhecimento suficiente para fazer o upload de novos aplicativos e atualizações.

Fique por Dentro

Será demonstrado como fazer o upload de um apk (Android Package) para o Developer Console e como publicar este aplicativo na loja oficial da Google. Para isso, será criado um pequeno app que fará uso do banco de dados SQLite, demonstrando todas as medidas e configurações necessárias (no Delphi e no console do desenvolvedor) para gerar um apk válido, e para que sua aplicação seja "aceita" e divulgada na maior vitrine de aplicações Android do mundo – a Google Play.

Android

É um sistema operacional baseado no núcleo do Linux, com diversas aplicações pré-instaladas. Desde que foi anunciado, o Android causou um grande impacto. Um dos motivos foi o envolvimento da Google e os recursos que a plataforma prometia oferecer, tais como: suporte a navegadores HTML, reprodução de vídeos pelo YouTube, Google Maps, armazenamento de dados e muito mais. Atualmente, o Android é bastante usado em dispositivos móveis como Smartphones, Tablets, TVs e já se especula muito sobre o que a Google vem chamando de Android vestível. Além dos dispositivos citados acima, o Android estará presente também em relógios, óculos e muito mais.

Por ter o código aberto, é bastante comum entre os fabricantes de equipamentos sua customização. Com menos de dez anos no mercado, já tomou proporções globais, tornando-se atualmente o sistema operacional móvel mais usado no mundo, e segundo a própria Google, a plataforma ganha um milhão de novos usuários todos os dias, estando presente em mais de um bilhão de dispositivos.

Open Handset Alliance

Grupo formado por empresas líderes em tecnologia móvel. Entre elas estão Samsung, Motorola, LG, Google e outras gigantes; essa aliança tem o objetivo de definir uma plataforma aberta e única, na tentativa de deixar os consumidores mais satisfeitos com seus produtos. Para isso, se faz necessário o uso de uma plataforma moderna e flexível para desenvolvimento de aplicações, surgindo assim o Android. Todos acabam se beneficiando dessa união: os fabricantes, os usuários e nós, desenvolvedores.

Google

Quando escutamos o nome Google já o associamos logo ao serviço de pesquisa mais usado e poderoso da web, mas essa empresa revolucionária possui muitos outros serviços igualmente bons, senão melhores como o Google Maps, o YouTube, o Gmail e o Tradutor.

E só existe o Android?

Na cola do Android estão outros dois gigantes da tecnologia: Apple, com o sistema operacional iOS e a Microsoft com o Windows Phone. Durante um bom tempo o iOS liderou o mercado Mobile, mas por ser um sistema operacional que só funciona em dispositivos fabricados pela própria Apple (iPhone e iPad), dispositivos estes muito bons – porém, com um preço razoavelmente alto – o iOS acabou sendo a preferência de usuários que têm um certo poder aquisitivo.

A Microsoft, por sua vez, líder absoluta quando assunto é Desktop, não vinha tendo o mesmo sucesso em dispositivos móveis, mas após o lançamento do Windows Phone 8 – que tem como um dos principais recursos a sincronização de arquivos entre o computador pessoal e o dispositivo do usuário – parece estar ganhando fôlego no mercado e de maneira rápida.

App

Com a explosão do mercado de dispositivos móveis, essa palavra passou a fazer parte de nosso vocabulário de maneira quase que imperceptível. App é a abreviação de Application, apelido carinhoso dado aos aplicativos de Smartphones e Tablets. Existem diversas categorias de App. As mais comuns são jogos, músicas e redes sociais. Para ter acesso a esses aplicativos, existem lojas online, nas quais empresas e desenvolvedores disponibilizam seus Apps para download (gratuito ou pago). Existem outras maneiras de fazer o download de um App para seu dispositivo, sem a necessidade de baixar diretamente das lojas oficiais, mas podem não ser seguros e até infectar seu dispositivo, causando danos muitas vezes irreparáveis.

Mas eu não sei Java!

Um dos pontos positivos da evolução das linguagens de programação é que esses avanços nos permitem trocar mais facilmente o papel de espectadores pelo de protagonistas.

Até pouco tempo atrás, desenvolver aplicativos móveis era exclusividade de desenvolvedores que conheciam Java, Object C, C# e etc. Com o lançamento das últimas versões do Delphi, mais especificamente XE4, XE5 e XE6, programadores adeptos da linguagem Object Pascal poderão desenvolver aplicativos para Android e iOS.

A novidade tem sido recebida com grande entusiasmo pela comunidade Delphi e está gerando muitas expectativas. Também não é pra menos, uma vez que a possibilidade de desenvolver aplicativos para Windows 32/64 bits, OS X, iOS e Android com o mesmo código e conhecimento é realmente impressionante.

Registro de desenvolvedores

Para você ter acesso ao Developer Console e poder distribuir seus aplicativos na Google play, é necessário fazer um registro, ou seja, criar uma conta de desenvolvedor. É bem simples e rápido, mas custa US$ 25. A justificativa que a empresa usa para cobrar pelo registro, é incentivar apps de alta qualidade e dificultar produtos com spam.

As exigências da Google, se comparadas à da sua concorrente Apple, são muito simples, visto que, para se registrar como desenvolvedor Apple Store é preciso pagar uma taxa anual, enfrentar uma burocracia imensa e aguardar um bom tempo; além disso, todos os apps são avaliados, podendo ser rejeitados e consequentemente não poderão ser disponibilizados para downloads.

APK - Android Package

Quando desenvolvemos aplicativos para a plataforma Windows e queremos instalá-los no Desktop de nossos clientes, é necessário levar o aplicativo (exe), uma série de dlls, banco de dados (no caso do servidor), além de arquivos de configuração. No desenvolvimento direcionado para Android, não precisamos nos preocupar com isso, pois todos os arquivos necessários para instalação do aplicativo no dispositivo estarão dentro do próprio apk, que corresponde ao executável. Desde as imagens até o banco de dados, é ele o único arquivo que deve ser enviado para a loja. A grosso modo, o apk é o seu app compilado e pronto para ser distribuído. Duas observações importantes a serem feitas é que o tamanho máximo suportado por um único apk é de 50 MB, a outra é que após instalado no dispositivo, o aplicativo terá um tamanho maior que o apk.

SQLite

Na maioria das vezes nossos aplicativos possuem persistência de dados. E para fazer uso deste recurso, precisamos de quê? Um banco de dados, é claro. No mundo Desktop existem inúmeros bancos para todos os tipos de projeto. São tantas opções que chegam a gerar polêmicas sobre qual é o melhor. No mundo Mobile a diversidade não é tanta, mas há algumas boas opções. A Embarcadero, por exemplo, está no mercado com sua versão Mobile do Interbase. Em nosso exemplo iremos utilizar o SQLite, por ser adotado pela maioria de desenvolvedores, ser compacto e nativo do Android, sem necessidade de nenhuma configuração. Além disto, vale ressaltar sua simplicidade (você não precisa ser DBA para dominá-lo) e ainda por cima é grátis. A maioria dos bancos relacionais são servidores independentes que se comunicam com as aplicações de forma cooperativa. O SQLite funciona de forma diferente, e, por isso, dizemos que é um banco integrado à aplicação, como se fosse uma biblioteca. Todas as operações são realizadas internamente, através de chamadas às funções da sua biblioteca. Na prática, acessamos os dados das tabelas através dos já conhecidos comandos (SELECT, INSERT, UPDATE e DELETE).

A seguir abordaremos as principais configurações que devem ser feitas para seu App sair do Desktop e ir direto para a Google Play.

No presente artigo estamos utilizando o Delphi XE6 (versão mais atual), mas é possível reproduzir o exemplo usando o XE5. Iremos tratar um pouco sobre os conceitos para em seguida aplicá-los. Como o objetivo não é mostrar a criação de um aplicativo em si, partiremos do ponto que você já tenha criado seu projeto e sua estrutura de pastas. Com o Delphi e o projeto aberto, veremos agora as principais configurações que devem ser feitas (a maioria requeridas) para seu apk ir direto para o Console.

Permissões

Quem já baixou um app da Google Play no seu Smartphone ou Tablet, deve ter percebido que antes da instalação ser iniciada, o Android nos envia uma mensagem (**Figura 1**), informando quais as permissões que o aplicativo está pedindo, ou seja, mostra quais os recursos a que o aplicativo precisa ter acesso livre para que possa funcionar corretamente. Cabe ao usuário aceitar ou não. Se aceitar, a instalação irá prosseguir, tendo acesso a todos os recursos informados; caso contrário, o aplicativo não será instalado, pois na Google Play não é possível liberar permissão para apenas um determinado recurso específico. Agora vem a pergunta que não quer calar: você lê a mensagem que o Android lhe envia informando as permissões que o aplicativo solicita?

Temos o costume de clicar (pelo menos a maioria) em aceitar sem ao menos ler, e acabamos "abrindo as portas" para qualquer um entrar e fazer o que quiser. É importante saber que ao liberar as permissões, estamos concordando que o aplicativo pode acessá-las a qualquer momento sem que nós possamos perceber. O que queremos dizer é que se for concedida esta permissão ao aplicativo para acessar os recursos de internet por exemplo, ele ficará livre para fazê-lo no momento em que precisar, consumindo seus créditos de recarga ou debitando em sua fatura.

Figura 1. Mensagem enviada pelo Android ao instalar qualquer aplicativo

Aplicativo de exemplo

Depois de entendermos todos os conceitos necessários, temos que fazer uso deste conhecimento. Iniciaremos então a criação do aplicativo, uma agenda telefônica. Uma app simples, mas que nos permitirá colocar em prática tudo o que foi visto. Sendo assim, iremos iniciar pelo banco de dados. Como citado anteriormente, a opção escolha foi o SQLite. Nosso banco também possuirá uma estrutura simples e de apenas uma tabela, conforme mostra a **Figura 2**.

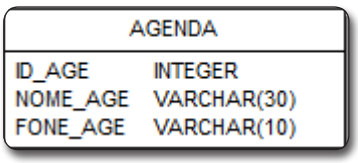

Figura 2. Demonstra a estrutura de nosso banco

Vamos dar início à criação da conexão com o banco. O procedimento é o mesmo usado para criar uma conexão com um banco Firebird por exemplo, porém, com uma diferença: não faremos uso de outra ferramenta para criar o banco (o arquivo em si), iremos utilizar o próprio Delphi para isso, mas nada impede o uso de uma ferramenta mais robusta que disponibilize mais recursos. Na paleta "Data Explorer", procuramos por "SQLite" e criamos uma nova conexão. Para o exemplo foi escolhido o nome de GooglePlay. Continuando, na janela "Modify Connection" (**Figura 3**) informamos onde o banco será salvo, e antes de confirmar as informações, damos uma passadinha em "Advanced", pois nesta janela é onde iremos informar se queremos que o Delphi apenas se conecte a um banco já existente ou se preferimos que ele crie um novo e se conecte ao mesmo.

No nosso exemplo iremos escolher a segunda opção (criar um novo banco e se conectar a ele). Para isso, alteramos a opção "FaillfMissing" (**Figura 4**) para "False" e confirmamos a escolha. Feitos esses procedimentos, já teremos nosso banco criado

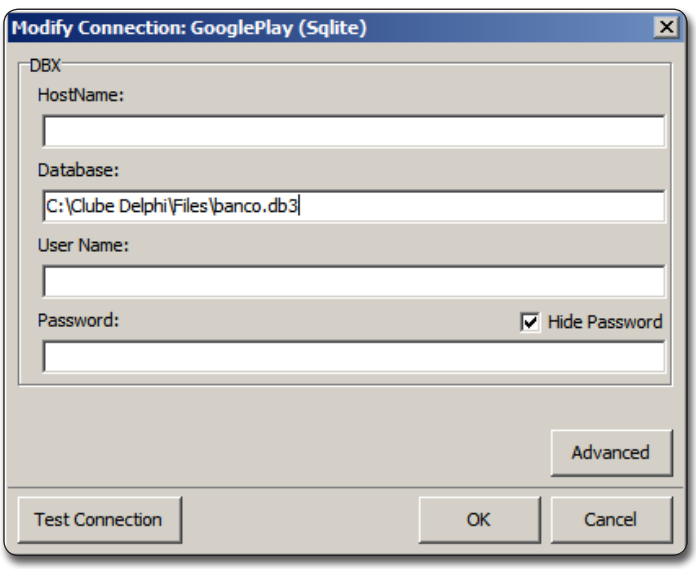

Figura 3. Janela onde é feita a conexão com o banco de dados

e nossa conexão realizada, restando apenas criar a tabela. Então localizamos no Data Explorer a conexão que acabamos de criar (GooglePlay) e abrimos o assistente de criação de tabelas através da opção new Table. Agora criamos a tabela com a estrutura demonstrada anteriormente na **Figura 2.**

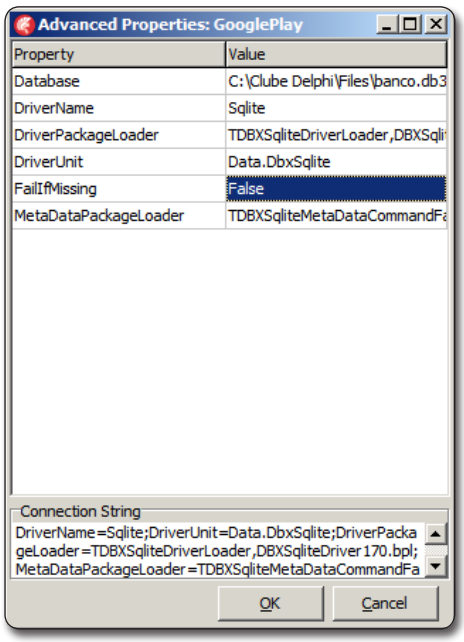

Figura 4. Desativando a propriedade FailIfMissing

Nota

É comum recebermos a mensagem de erro "sqlite3.dll not found", quando se adiciona uma nova conexão com SQLite. Neste caso, o Delphi nos informa que não encontra a dll. Para resolver o problema, faça o download da dll (seção **Links**) e a salve na pasta System32 ou SysWOW64, dependendo da arquitetura de seu sistema operacional.

Agora é a vez da interface do usuário. Aqui não abordaremos esse assunto, mas ele tem grande importância em um aplicativo influenciando diretamente na decisão de um usuário realizar ou não o download. Vamos precisar de um TabControl com alinhamento Client e duas TabItem com os nomes de tbLista e tbCadastro e a propriedade Text preenchida com os textos "Lista" e "Cadastro". Adicionamos na tbLista uma ToolBar e um ListBox com alinhamento Client, nome lbList e a propriedade com ItemStyle listboxitemrightdetail. Por último, arrastamos um SpeedButton para a ToolBar e lhe damos o nome de "sbSair". Neste momento, o aplicativo terá layout conforme mostrado na **Figura 5**.

Na tbCadastro colocamos uma ToolBar e dentro dela arrastamos dois SpeedButtons, com os nomes "sbAdicionar" e "sbLista". Configuramos a propriedade StyLookup para "donetoolbutton" e "searchtoolbutton", respectivamente. Adicionamos também dois Labels e dois Edits. Assim nossa App estará conforme na **Figura 6**.

Partiremos agora para a gravação do contato no banco de dados. Existem várias maneiras para se realizar este passo, porém, abordaremos a mais prática. Arrastamos a tabela Agenda presente no "Data Explorer" para a tbCadastro. Ao fazer isso, será criado um SQLConnection (GooglePlayConnection) e um SQLDataset (AgendaTable). Alteramos a propriedade "Connected" do SQLConnection para "True" e a propriedade "Active" do SQLDataSet também para "True". Arrastamos outro SQLDataSet para tbCadastro e lhe damos o nome de "sqlAdicionar".

Figura 5. Interface da tbLista

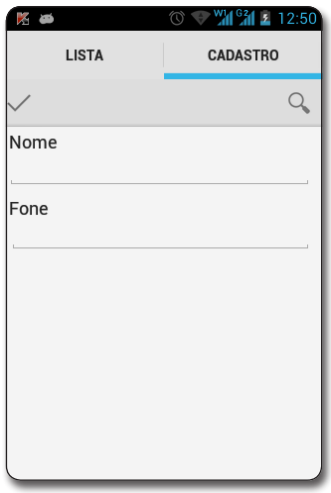

Figura 6. Interface da tbCadastro

Por fim, apontamos a propriedade SQLConnection do sqlAdicionar para o componente GoogleplayConnection. Feito isso termos todos os componentes de acesso a dados configurados e já podemos adicionar o evento OnClick do botão sbAdicionar, conforme a **Listagem 1**.

Listagem 1. Algoritmo do botão Adicionar

```
01.  procedure TFormPrincipal.sbAdicionarClick(Sender: TObject);
02.  var
03.    sSqlInserir : string;
04.  begin
05.    if  edtNome.Text.IsEmpty or edtFone.Text.IsEmpty then
06.      ShowMessage('Todos os campos são obrigatórios')
07.    else
08.    begin
09.      sSqlInserir := ' INSERT INTO AGENDA(ID_NOME, ID_FONE) ' +
10.                 ' VALUES ' +
11.                 ' (' + edtNome.Text.QuotedString + ',' +
12.                 ' ' + edtFone.Text.QuotedString + ')';
13.     try
14.       sqlAdicionar.CommandText := sSqlInserir;
15.       sqlAdicionar.ExecSQL;
16.       ShowMessage('Contato Salvo!');
17.       AgendaTable.Close;
18.       AgendaTable.Open;
19.       edtNome.Text :='';
20.       edtFone.Text :='';
21.       tabPrincipal.ActiveTab := tbLista;
22.     except
23.       on E: Exception do
24.         ShowMessage(E.Message);
25.     end;
26.   end;
27.  end;
```

Observe no código que primeiramente é verificado o preenchimento dos Edits edtNome e edtFone (linha 05), que, caso não sejam preenchidos corretamente, exibirão uma mensagem de obrigatoriedade (linha 06). A seguir, entre as linhas 09 e 12 é montada a instrução de inserção da tabela AGENDA. Nas linhas 14 e 15, a instrução é atribuída ao componente sqlAdicionar, sendo executada imediatamente. Posteriormente, temos a mensagem de que o contato foi salvo, a abertura da tabela AGENDA, exibindo os dados e a limpeza dos Edits (linhas 17 a 20). Em caso de erro, uma mensagem é mostrada e a exceção capturada (linhas 23 e 24).

A **Listagem 2** ilustra os eventos OnClick para os componentes "sbLista" e "sbsair". Note que ao clicar na lista, é selecionada a página referente (linha 03). Já na linha 07 é utilizado o comando MainActivity.Finish. Activity, em Android, explicando de forma simples, é apenas uma tela, a interface do usuário. No Delphi corresponde de forma análoga aos nossos Forms. Como no exemplo só há uma Activity, ou seja, um formulário, ao usarmos o método Finish, iremos tirar nossa App da pilha, isto é, encerraremos o aplicativo.

A interface de cadastro já está finalizada, o próximo passo será a lista. Assim, temos que fazer com que todos os contatos cadastrados no banco de dados possam ser visualizados no lbList. Para que o Listbox mostre os contatos com seus respectivos telefones cadastrados, deve-se fazer uso do LiveBindings, ligando o campo nome à propriedade Item.Text do lbList e o campo fone à propriedade Item.Detail do mesmo componente. A **Figura 7** demonstra como deve ser feita essa ligação.

Listagem 2. Algoritmo dos botões Lista e Sair

```
01.  procedure TFormPrincipal.sbListaClick(Sender:TObject);
02.  begin
03.    tabPrincipal.ActiveTab := tbLista;
04.  end;
05.  procedure TFormPrincipal.sbSairClick(Sender:TObject);
06.  begin
07.    MainActivity.finish;
08.  end;
```

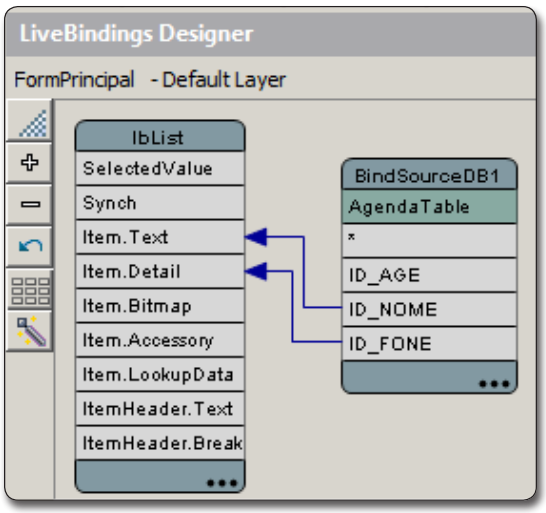

Figura 7. Ligação dos campos com o ListBox

Para finalizar as configurações que envolvem banco de dados, ainda nos restam duas tarefas importantíssimas. A primeira delas é o ajuste do Deployment, ou seja, configurar o "envio" do arquivo banco.db3 para o dispositivo. Para isso, temos que informar onde o arquivo está localizado no momento e qual será o seu destino, isto é, em qual pasta do dispositivo ele será salvo. Podemos fazer esta configuração no menu Project>Deployment, nesta janela (**Figura 8**), os dois campos que nos interessam são Local Path que indica onde o nosso banco está atualmente e Remote Path, que indica onde o banco será salvo. No caso do Android sempre será em Internal. A segunda etapa é especificar para o SQLConnection a localização do banco de dados no dispositivo móvel, deste modo, esta informação é passada no evento BeforeConnect do mesmo, conforme mostra a **Listagem 3**.

Finalizados estes passos, já é possível cadastrar alguns contatos e observar a lista sendo preenchida (**Figura 9**).

Assim concluímos as configurações referentes a banco de dados e podemos seguir em frente. A "grande funcionalidade" de nosso aplicativo será a possibilidade de clicar em um contato da lista e dali mesmo efetuar uma chamada telefônica, utilizando o discador nativo do Android. O código que fará isso pode ser observado na **Listagem 4**.

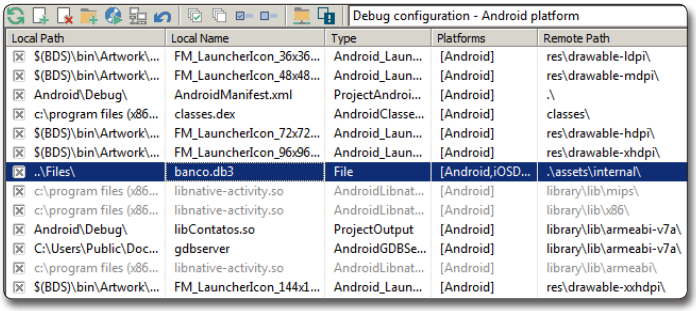

Figura 8. Configuração do Deployment

Figura 9. Ligação de contatos cadastrados no banco de dados

Listagem 3. Algoritmo do SQLConection

```
01.  procedure TFormPrincipal.GoogleplayConnectionBeforeConnect(Sender:
     TObject);
02.  begin
03.    {$IFDEF ANDROID}
04.      GoogleplayConnection.Params.Values['Database'] := GetHomePath +
         PathDelim + 'banco.db3';
05.  {$ENDIF}
06.  end;
```

Listagem 4. Algoritmo para acessar o recurso do discador

```
01. private
02. { Private declarations }
03. Phone : IFMXPhoneDialerService;
04. procedure TFormPrincipal.lbListItemClick(const Sender:TCustomListBox;
    const Item:TListBoxItem);
05. begin
06.  TPlatformServices.Current.SupportsPlatformService(
     IFMXPhoneDialerService, IInterface(Phone));
07.   Phone.Call(Item.ItemData.Detail);
08. end;
```

Observe que devemos informar para qual número queremos telefonar, no nosso casso passamos a propriedade ItemData.Detail do ListBox (linha 07), pois é ela quem possui esta informação. Agora se o usuário deseja ligar para qualquer contato, precisará de apenas um Click, sem a necessidade de decorar o número, sair do aplicativo e ir digitá-lo novamente. O objetivo é dar o maior número de opções para o usuário, lembre-se que serão muitos os aplicativos disputando a atenção com o nosso. Aqui foi passado um exemplo de como efetuar ligações, porém, também poderíamos enviar e-mails, SMS, compartilhamentos no Facebook e etc.

Nota

É preciso adicionar as units FMX.PhoneDialer e FMX.Platform ao uses do formulário.

Um pouco mais sobre permissões

Vimos anteriormente que para que o nosso aplicativo tenha acesso a determinado recurso do dispositivo, é preciso que seja informada a permissão necessária. Como foi mostrado na **Listagem 4** acabamos de fazer um uso de um recurso do dispositivo (o discador), então, devemos informar que o nosso App precisa dessa permissão, nesse caso seria a "Call Phone".

Sendo assim, veremos como configurar essas permissões em nosso projeto. Como sempre, o Delphi procura facilitar a vida do desenvolvedor ao máximo, então, para informar as permissões que o aplicativo necessita, teremos uma simples janela (**Figura 10**) onde serão selecionadas as permissões necessárias.

Apesar de simples, devemos ficar atentos, pois, caso não se informe a permissão que o aplicativo necessita, ele não irá funcionar corretamente no dispositivo do usuário. Mas veja bem, não habilite todas as opções. Uma permissão indevida pode fazer com que o usuário deixe de baixar seu aplicativo. Lembre-se que habilitar vários recursos que não estão explícitos no seu aplicativo, pode torna-lo sob suspeita de ser malicioso. Sabendo disso, solicitamos apenas permissões que nosso App irá realmente precisar.

Já sabemos que é muito importante informar todas as permissões que o App precisa e que não podemos sair marcando qualquer uma, porém, é necessário ter conhecimento do que se deve habilitar.

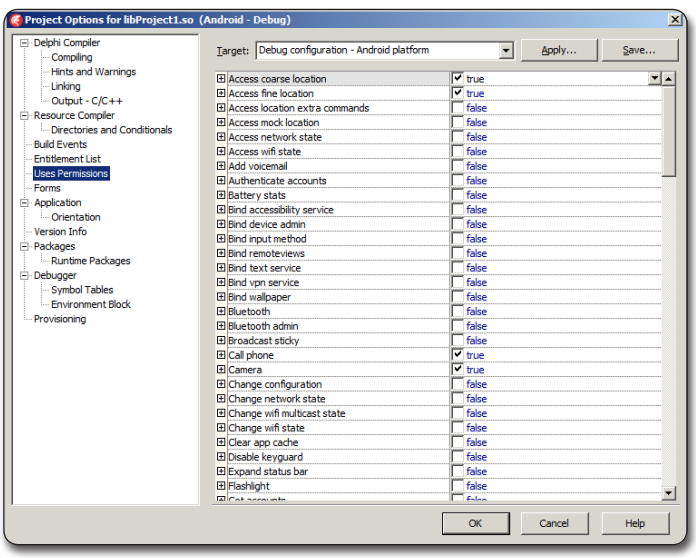

Figura 10. Ligação de contatos cadastrados no banco de dados

Nota

É obrigatório informar as permissões necessárias, porém, caso você deixe de informar alguma, de todo modo o seu apk será enviado para o Console de desenvolvedor, e consequentemente, para a Google Play, no entanto quando o usuário baixá-lo e executá-lo, ele não irá funcionar.

O Android possui uma documentação completa que abrange bem a parte de permissões e sua leitura é altamente recomendada (veja mais na seção **Links**). Adiantamos aqui quatro delas, apenas para você fixar melhor o conceito:

• CALL_PHONE: permite ao aplicativo iniciar uma chamada telefônica;
• SEND_SMS: permite ao aplicativo enviar SMS;
• REBOOT: permite ao aplicativo o poder de reiniciar o dispositivo;
• BATTERY_STATS: permite ao aplicativo coletar informações sobre o estado da bateria do dispositivo;

Nome e versão do pacote

O nome do pacote e o número da versão é outra configuração que não pode passar despercebida. Elas influenciam tanto no primeiro envio da aplicação para o Developer Console, como em atualizações do aplicativo. A escolha do nome do pacote deve ser levada a sério e bem planejada, pois o nome é exclusivo e permanente, ou seja, depois de criado não poderá ser reutilizado ou excluído.

É recomendado que o nome do pacote seja formado pelo seu domínio (ao contrário) concatenado ao nome da sua aplicação, tornando-o mais exclusivo possível. Digamos que o domínio é suaempresa.com.br e sua App chama-se "meuscontatos".

O nome do pacote ficaria br.com.suaempresa.meuscontatos. A escolha do nome é um processo importante até mesmo para facilitar a localização da App na loja pelo público.

Na **Figura 11** podemos observar os campos package e "Version Name" preenchidos conforme o exemplo citado.

Até o momento foi abordado apenas do nome, mas a versão também tem um papel importante. No primeiro upload do apk para o Developer Console, a versão deve ser 1 (**Figura 11**) e sempre que o aplicativo sofrer uma atualização, deve-se incrementá-lo em 1. Por exemplo, digamos que implementamos o recurso de envio de SMS para os contatos e desejamos disponibilizar essa nova funcionalidade para os usuários do aplicativo, é preciso obrigatoriamente mudar a Version code para 2. Assim os usuários que possuem a versão antiga terão seus dispositivos atualizados de forma automática, logo após publicar a versão na loja.

Nota

O nome do pacote atualizado deve ser o mesmo da versão atual, o código da versão precisa ser superior ao da versão atual e o apk precisa ser marcado com a mesma assinatura da versão atual (veremos mais adiante).

Ícone do Aplicativo

Alterar o ícone do aplicativo não é obrigatório, entretanto, é aconselhável que seja feito, pois assim o App ganha uma identidade junto ao usuário. Além disso, é esse mesmo ícone que aparecerá no dispositivo.

Devido à grande diversidade de dispositivos e resoluções de tela, é necessário informar um ícone para cada tipo de resolução (ldpi,mdpi,hdpi,xhdpi,xxhdpi).Só pra deixar claro, não é uma imagem para cada resolução, e sim a "mesma" imagem com resoluções diferentes. Após a escolha de uma imagem de nossa preferência, ela deve ter os tamanhos de 36x36, 48x48, 72x72, 96x96 e 144x144. Existem vários programas gratuitos que podem fazer o redimensionamento de sua imagem. Se não alterar o ícone da aplicação, ela irá com o ícone padrão do FireMonkey. Veja na **Figura 12** onde é modificado o ícone para as cinco resoluções citadas anteriormente.

Nota

A imagem deve ser no formato png.

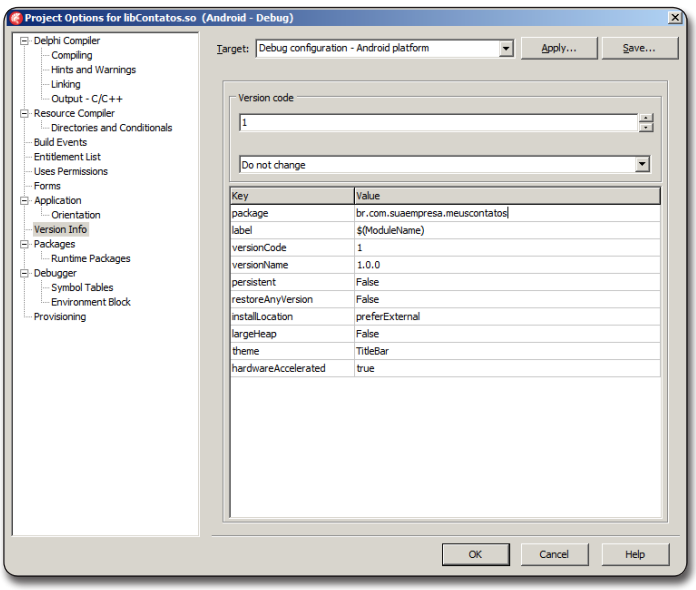

Figura 11. Tela onde informamos o nome e versão do pacote

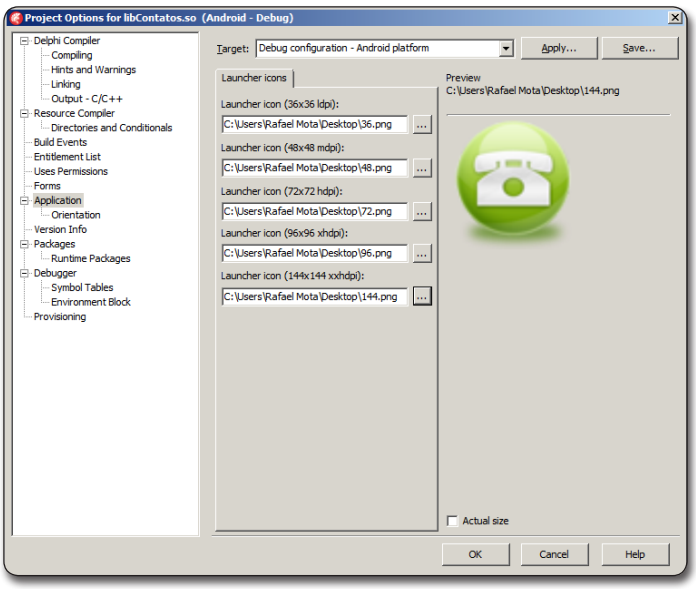

Figura 12. Local onde alteramos o ícone da App

Certificado digital

Para que um aplicativo seja instalado em um dispositivo – seja ele Smartphone ou Tablet –, o aplicativo deve ser assinado usando um certificado digital. Este certificado pode ser criado por você mesmo, sem a necessidade de uma Certificadora Autorizada. Para isso, precisamos apenas gerar uma chave privada. Esta chave é apenas um arquivo com a extensão ".keystore" que você usará para assinar seu aplicativo. A **Figura 13** demonstra onde é criada a Keystore.

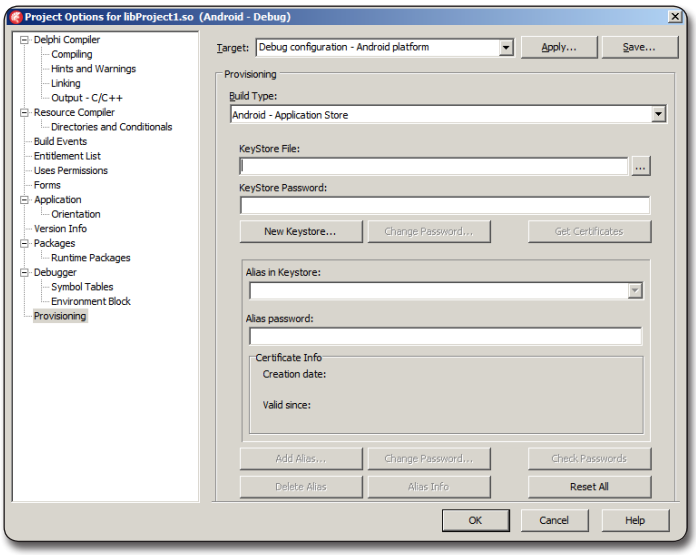

Figura 13. Janela onde criamos a Keystore

O Delphi mais uma vez simplifica bastante o nosso trabalho. Basta clicar no botão "New Keystore" e será exibido o Wizard de criação (**Figura 14**). Nesta tela escolhemos onde a chave será salva por meio do campo "File" e definimos uma senha no campo Password. Feitos esses procedimentos já teremos a nossa chave privada, faltando agora apenas assinar a aplicação.

Assim que concluímos os passos anteriores, o Wizard nos levará à outra janela (**Figura 15**). Nesta nova tela devemos informar um alias para a assinatura, confirmar a senha informada anteriormente, e por fim, alguns dados como nosso primeiro e último nome.

Feito isso, nossa aplicação está assinada digitalmente e quase pronta para ser enviada ao console do desenvolvedor. Compare a diferença entre as **Figuras 13** e **16**. Na primeira ainda não foi gerada a chave privada (keystore) e, consequentemente, ainda não foi criada a nossa assinatura. Já na **Figura 16** o processo está concluído.

A assinatura do aplicativo é uma das configurações mais importantes, senão a mais importante, pois, é ela quem comprova quem é o desenvolvedor do aplicativo.

É importante entender que quando estamos fazendo o Deploy da aplicação do Delphi diretamente para um dispositivo móvel ou para o emulador, o próprio IDE gera a Keystore e assina nossa App de forma automática, no entanto, quando se trata de gerar o arquivo para a Google Play, nós precisamos fazê-lo.

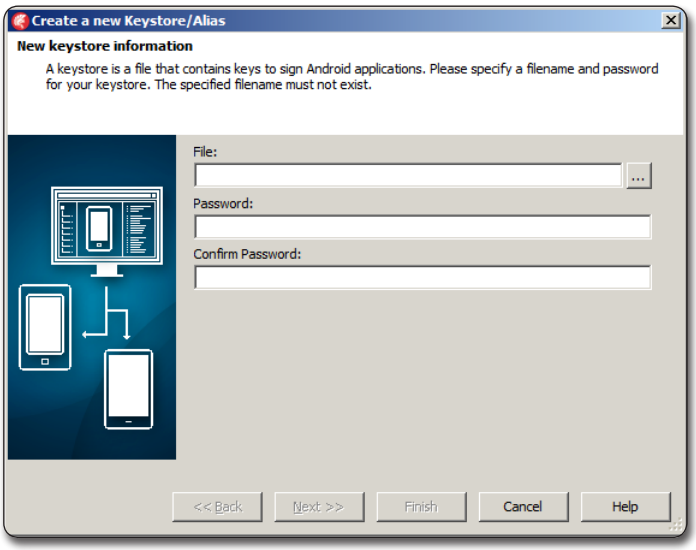

Figura 14. Wizard para criação da Keystore

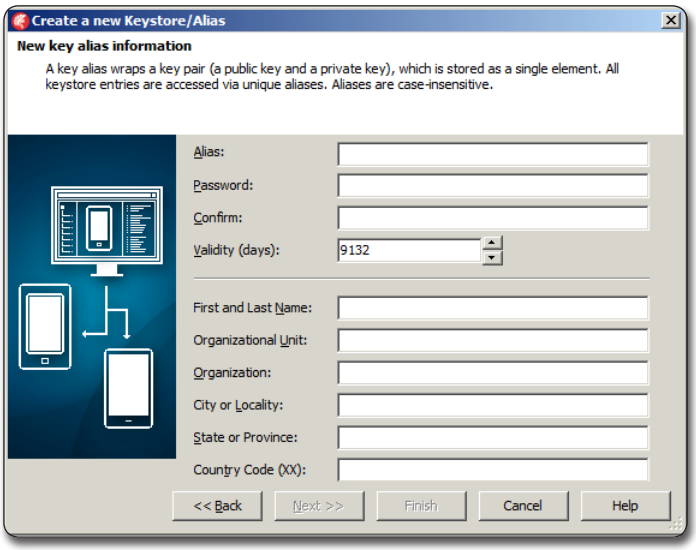

Figura 15. Segunda tela do Wizard

Nota

Ao enviar seu app pela primeira vez para o Console, somos obrigados a gerar a Keystore. Ela também é peça fundamental nas atualizações de aplicação, pois, sempre que é enviada uma nova versão para a loja, é verificado se a Keystore é a mesma da versão atual.

Gerando o apk

O último passo a ser feito é gerar o apk. Podemos estar pensando: mas sempre que compilamos o projeto, ele não gera o apk? A resposta é sim, mas ele cria em um modo que o console não irá aceitar, que é o modo de depuração. Ao tentar enviar um apk em modo de depuração, o console exibirá a mensagem mostrada na **Figura 17**.

Para compilar um apk de maneira correta, contamos com a ajuda do Project Manager, bastando apenas acessar as configurações da plataforma Android e alterar a opção de "Debug" para

"Application Store", com um simples duplo clique, conforme mostra a **Figura 18**.

Pronto! Essa foi a última configuração necessária no Delphi. O apk já está pronto para ser levado ao console.

Registrando-se como desenvolvedor

Abordaremos o processo de se registrar como desenvolvedor, para que seja possível o acesso ao Console do Desenvolvedor, local onde gerenciamos todos os seus aplicativos e os enviamos para a Google Play.

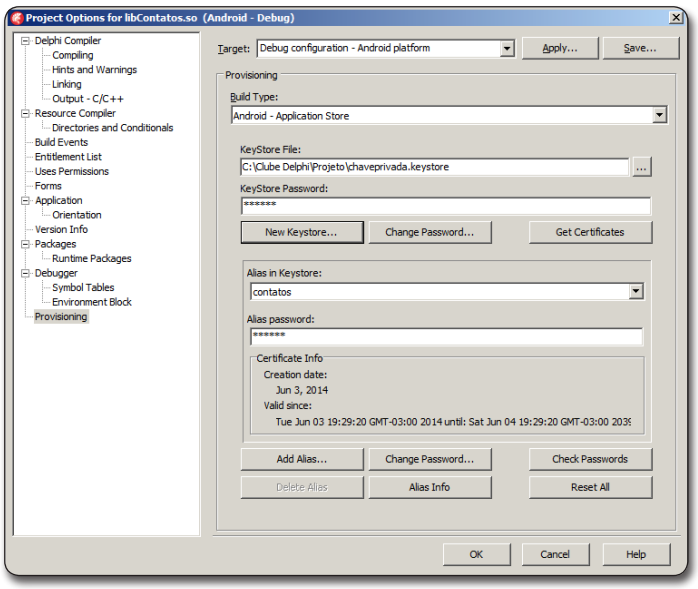

Figura 16. Tela demonstrando que o nosso aplicativo já está assinado digitalmente

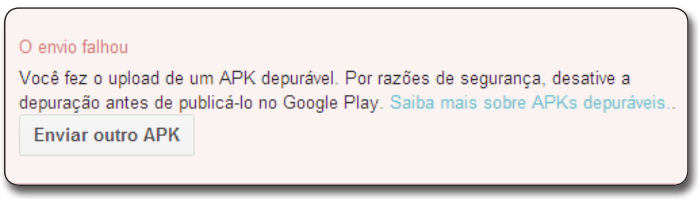

Figura 17. Demonstra o erro que o console irá lhe retornar se tentar fazer o upload de um apk no modo de depuração

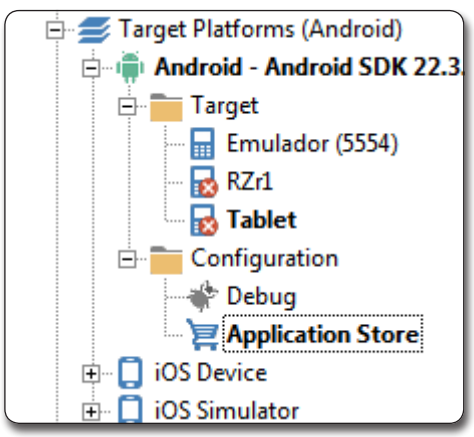

Figura 18. Local onde configuramos o modo de compilação para Application Store

O primeiro passo antes de iniciar o cadastro é criar uma conta no Gmail. Após a criação da conta, acessaremos o formulário de cadastro (veja seção **Links**). Este formulário é dividido em quatro etapas. Na primeira etapa deve-se informar qual conta do Gmail será associada ao console; a segunda etapa é onde devemos aceitar o contrato do Desenvolvedor.

Antes de prosseguir para a próxima etapa, é importante ler os termos do contrato e as condições de uso, pois, neles encontramos informações importantes como locais (países) para os quais podemos distribuir aplicativos. É necessário também ter um cartão de crédito em mãos. Clicamos no botão "Continuar pagamento" para continuar.

Na terceira etapa devemos informar nome e local de residência e também os dados do cartão (número, vencimento e etc.). Em seguida, clicamos no botão "Aceitar e continuar", conferimos as informações e clicamos no botão "Comprar". Se tudo ocorrer bem, estaremos na quarta e última etapa na qual deve-se informar o perfil de desenvolvedor, ou seja, nome, e-mail, site e etc. Vale lembrar que esses dados são os que apareceram na Google Play para que os outros usuários vejam. Basta clicar no botão "Registro Completo".

Concluído o processo, possuímos uma licença de desenvolvedor e já estamos aptos a enviar apks para o Console, publicando aplicativos desenvolvidos.

Publicando um aplicativo

Agora já estamos bem próximos de publicar nossa primeira app na Google Play, faltando apenas fazer o upload do nosso apk para o console e fazer as últimas configurações.

O Console de desenvolvedor é um pouco extenso para ser explicado por completo em apenas um artigo. Então iremos abordar apenas o necessário para publicar sua aplicação.

Acessamos o console (endereço disponível na seção **Links**). A primeira tela que vemos (**Figura 19**) mostrará todos os aplicativos que já estão publicados, caso existam. Para iniciar o upload do apk, selecionamos a opção "Adicionar novo Aplicativo" e seguimos os passos que o Console irá solicitando. Um deles é informar o local no qual se encontra o apk. O Delphi por padrão cria uma pasta chamada "Android" no diretório onde foi salvo o projeto. Dentro dessa pasta, haverá outra chamada "Release". Nessa por sua vez, conterá o diretório "Bin", e é nela que se encontrará o apk. Se tudo estiver certo com o apk, o upload será realizado.

Realizado o upload, devemos fazer uma visita a algumas abas do console (**Figura 20**). Em cada uma das abas teremos que fazer configurações para que o aplicativo seja liberado para publicação na loja. Durante este processo podemos consultar o botão "Rascunho" (**Figura 21**), pois, enquanto ele estiver com o nome de exibição "Rascunho", o aplicativo ainda não poderá ser publicado, indicando que ainda faltam configurações a serem feitas. Para que saibamos exatamente quais são essas configurações, basta selecionar a opção "Por que não posso publicar?" da **Figura 21**.

Explorando as opções da aba "Detalhes do aplicativo" (**Figura 20**), vamos comentar sobre os itens "Recursos Gráficos" e "Categorização".

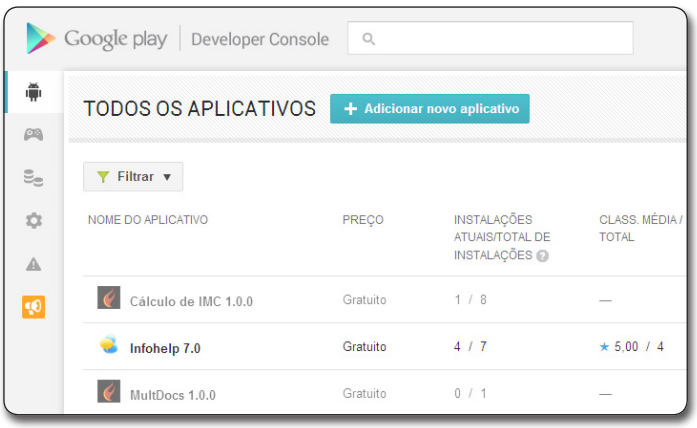

Figura 19. Tela principal do Console

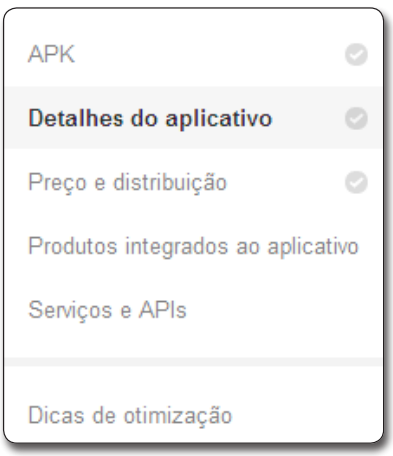

Figura 20. Abas de configuração do aplicativo

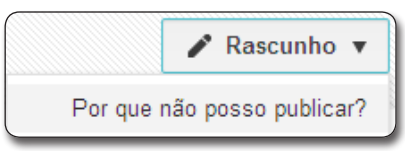

Figura 21. Botão indicando que sua aplicação está em modo rascunho e não pode ser publicada

No primeiro devemos fazer capturas de telas do aplicativo funcionando em dispositivos (Tablets (7 e 10 polegadas) e Smartphones). Sugiro que escolhemos bem essas imagens, pois será a apresentação do aplicativo para o usuário. O ideal é procurar selecionar as telas que possuam mais recursos e chamem mais atenção. Caso não possua diversos dispositivos para rodar o aplicativo, as capturas de telas podem ser feitas apenas no dispositivo que possuímos, porém, os usuários de outros dispositivos, como Tablets por exemplo, não terão noção de como o aplicativo se comportará em telas maiores.

O item "Categorização" tem uma importância muito grande, pois, aqui é informado se o aplicativo é um jogo ou uma App comum, qual a sua categoria (Entretenimento, Esportes, Humor e etc.). Vale lembrar que a escolha da categoria afeta diretamente nas buscas que os usuários fazem por aplicativos. Os outros itens desta aba são bem dedutíveis, então não serão detalhados neste artigo.

A próxima aba é "Preço e Distribuição". Essa guia é a mais simples de todas, no entanto, exige certa cautela. Aqui deve-se escolher em quais países que se deseja distribuir o aplicativo, aceitar alguns termos e etc. É aqui também que informamos se o aplicativo será pago ou gratuito. Parece ser uma escolha simples, mas saiba que ao informar que o aplicativo será gratuito, futuramente não poderá alterá-lo para pago.

Por fim, a aba "Dicas de Otimização". Como o próprio nome diz, aqui serão abordadas dicas para otimizar o aplicativo. Exemplo: habilitá-lo para múltiplos idiomas.

Depois de realizar todas as configurações, o botão citado anteriormente na **Figura 21**, irá mudar de nome exibindo a mensagem "Pronto para publicar", conforme a **Figura 22**. Após aguardar algumas horas, seu aplicativo estará disponível para download para o mundo inteiro.

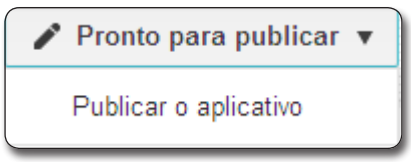

Figura 22. Botão indicando que sua aplicação já pode ser publicada

Agora que fomos apresentados mais intimamente ao Android, o sistema operacional em maior evidência na atualidade, já é possível iniciar a jornada no mundo Mobile.

Autor

Rafael Mota
inforafael.mota@gmail.com
Desenvolvedor Delphi, estudande da linguagem Java, plataforma Android e um apaixonado por tecnologia.

Links:

Criar conta no Gmail
https://accounts.google.com/SignUp

Formulário de conta e acesso ao Console
https://play.google.com/apps/publish/signup/

Permissões:
http://developer.android.com/reference/android/Manifest.permission.html

Baixar sqlite3.dll
http://www.sqlite.org/download.html

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DataSnap: Migrando do dbExpress para FireDAC

Fique por Dentro

O presente artigo tem como objetivo mostrar o uso da mais nova biblioteca de acesso a dados do Delphi, o FireDAC, em projetos multicamadas DataSnap. Logo, a intenção aqui é induzir o desenvolvedor, de forma natural, a tirar proveito do que há de mais novo no cenário da ferramenta. Para isso, serão mostradas práticas já conhecidas deste tipo de desenvolvimento, baseadas tradicionalmente no dbExpress, que até então era tido como a principal opção nativa para acesso a dados no processo construtivo. Adicionalmente, outra motivação pela escolha do tema se dá pela crescente adoção, por parte da comunidade, deste segmento de projeto (DataSnap). A justificativa para isso incide diretamente sobre diversos fatores, cujos principais são: independência de tecnologia de banco de dados, escalabilidade, organização de código, robustez arquitetural, centralização das regras de negócio e clientes leves. Para este último, destaque para as recentes aplicações nativas Mobile (Android e iOS).

Em suma, dbExpress e FireDAC podem ser definidos como sendo frameworks de acesso a dados que atualmente integram a suíte de recursos do Delphi. Na mais recente versão da ferramenta, a XE6, ambas são naturalmente tidas como as principais opções, em termos de componentes, para a construção de aplicações que envolvam banco de dados. Neste sentido, historicamente, dois tipos de projetos se destacam. Um deles está relacionado ao provimento de aplicações tradicionais de banco de dados, enquanto que o outro também acaba por englobar aplicações de banco de dados, porém dentro de um contexto multicamadas, através do uso da tecnologia DataSnap.

dbExpress

De forma abrangente, o dbExpress é uma tecnologia surgida ainda na versão 6 do Delphi, que contemplava uma série de elementos destinados à conectividade e acesso a servidores de bancos de dados relacionais. Em suma, tais elementos se refletiam em classes, drivers e componentes que, na prática, contribuíam de forma efetiva no desenvolvimento de aplicações de banco de dados com a ferramenta. Inicialmente, toda a fundação do framework foi escrita em linguagens mais tradicionais da época, como C e C++, contudo, no Delphi 2007 todo este cenário é alterado, mediante uma total reescrita do framework para a Delphi Language, tornando-o essencialmente nativo. Seguindo por esta evolução, no release seguinte do IDE (Delphi 2009) o dbExpress passa a, efetivamente, fazer parte do contexto DataSnap, a partir do provimento de um driver cliente exclusivo para a tecnologia. A partir deste momento, o dbExpress acabava por contemplar não somente servidores de banco de dados, mas também servidores de aplicação. Na prática, os desenvolvedores estavam então aptos a conectar suas aplicações clientes nativas a Servidores DataSnap, por meio do uso de um tradicional TSQLConnection associado ao referido driver.

Por fatores como este, o uso do dbExpress em projetos DataSnap envolvendo dados, passou a ser algo cada vez mais natural no cotidiano de boa parte da comunidade, que prezava pela adoção de soluções nativas. Em contribuição a este prognóstico, está a própria característica dos DataSets da biblioteca do dbExpress – TSQLDataSet, TSQLQuery e TSQLTable – que são tidos como unidirecionais e, portanto, não suportam aspectos como cache de dados e navegabilidade plena entre os registros. Logo, sua efetiva utilização fica condicionada à utilização de elementos do tipo TDataSetProvider e TClientDataSet que, acabam por contribuir de forma positiva ao cenário multicamadas de projetos DataSnap. Isto porque, uma vez que os dados são obtidos a partir do Servidor para a aplicação Cliente, os mesmos são mantidos em memória local, não exigindo uma conexão direta, ativa e permanente com o referido servidor.

Anteriormente aos recentes releases do IDE (vide XE5 e XE6), o dbExpress ocupava, de forma soberana, o posto de "principal opção nativa para acesso a dados". Já no cenário atual do Delphi, ainda sob o aspecto nativo, o dbExpress continua sendo peça chave para a conectividade cliente em projetos DataSnap, todavia, sua posição relacionada a acesso a dados foi totalmente alterada, sendo preterida pela nova tecnologia denominada FireDAC.

FireDAC

No Delphi, o FireDAC envolve um conjunto de componentes de acesso a dados tidos como "Universal Data Access Components", direcionados ao desenvolvimento de aplicações de banco de dados cross-platform. Logo, ele se mostra útil na construção

de projetos para as duas principais plataformas suportadas pela ferramenta atualmente, que inclui a tradicional Desktop, por meio dos sistemas operacionais Microsoft Windows e Mac OS, bem como a recente Mobile, refletida nos ambientes móveis Android e iOS. Em decorrência disso, diversos são também os tipos de dispositivos atendidos neste cenário, pontualmente PCs, Tablets e Smartphones.

Mediante sua singular arquitetura, o FireDAC provê acesso direto, nativo e de alta performance a uma extensa gama de SGBDs distintos. Na versão mais recente do Delphi, a XE6, a listagem oficial inclui nominalmente 13 nomes, que incluem fontes intermediárias ODBC, servidores de banco de dados e bases locais:

- Advantage Database Server;
- Sybase SQL Anywhere;
- IBM DB2 Server;
- Firebird;
- Embarcadero InterBase;
- Informix;
- Microsoft Access Database;
- Microsoft SQL Server;
- MySQL;
- ODBC;
- Oracle;
- PostgreSQL;
- SQLite.

Em complemento a tudo isso estão alguns pontos de uso que acabam por fundamentar todo o aspecto técnico e comercial que atualmente colocam o FireDAC como sendo a principal opção nativa para lidar com a parte de acesso a dados de uma aplicação Delphi. Como demonstrativo, a seguir são enumeradas algumas dessas nuances, apresentadas de forma oficial pela Embarcadero, como forma de incentivo à utilização do produto:

- Extenso "know-how", proveniente do AnyDAC (seu produto co-irmão), fruto de mais de uma década de experiência no desenvolvimento de aplicações reais de banco de dados;
- Abstração de todas as complexidades existentes na interação da aplicação com o banco de dados. Como consequência positiva, o desenvolvedor tende a se concentrar somente no desenvolvimento efetivo e regras de negócio;
- Provimento de recursos concretos, suficientes a atender as particularidades relacionadas a cada SGDB;
- Provimento de componentes dos mais variados tipos, voltados aos mais variados fins, que abrangem desde operações triviais, tais como conexão com o banco de dados e recuperação/atualização de dados, até serviços especializados, que incluem Backup, Restore, Trace, entre outros;
- Custo reduzido de propriedade, uma vez que é distribuído de forma nativa nas principais edições do Delphi;
- Dada sua similaridade com o dbExpress, tanto em termos de funcionalidade quanto em termos de uso, sua adoção é bastante natural e facilitada, o que contribui para uma rápida familiarização, por parte do desenvolvedor.

DataSnap

Anteriormente alcunhado como MIDAS, o DataSnap é uma tecnologia presente no Delphi há mais de uma década, e que possibilita o desenvolvimento de aplicações multicamadas, tradicionalmente conhecidas também como "aplicações multi-tier". Tal denominação se dá em função da segmentação do âmbito geral da aplicação em partes bem definidas, cada qual tratada então como "camada". No cenário do desenvolvimento, esse tipo de abordagem se mostra bastante tradicional, sendo utilizada de forma efetiva em construções de aplicações nas mais variadas tecnologias e linguagens, tais como C# e Java, só para citar duas. Além disso, a fundamentação por trás do conceito da arquitetura de uma aplicação multicamadas não especifica ou determina um número exato de partes (ou camadas) que o projeto deve ter. Logo, isso fica a cargo da proposta relacionada, atrelada às facilidades e recursos que a tecnologia/linguagem utilizada tem a oferecer. Especificamente no Delphi, tal quantidade é essencialmente definida em três, tal como mostra a **Figura 1**.

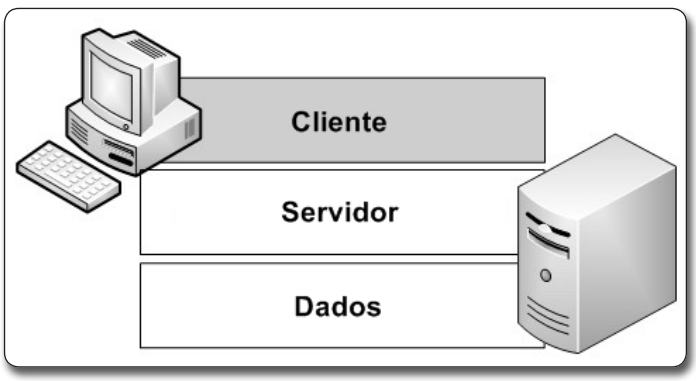

Figura 1. DataSnap – Camadas

Logo, o ponto chave desta arquitetura se dá pela presença de uma camada intermediária, figurada numa estrutura denominada Servidor de Aplicação, que fará a interconexão entre a camada cliente e a camada de dados. Sobre esta última, ela acaba por fundamentar outra inclinação deste tipo de abordagem no cenário da ferramenta, uma vez que se refere à presença de um banco de dados na conjuntura da aplicação. Isto porque, em termos gerais, o conceito multi-tier não necessariamente está associado a aplicações de banco de dados, todavia, no Delphi isso pode ser considerado uma verdade. A fim de se estabelecer conceitos, a seguir é feita uma breve explanação sobre cada "tier" mencionada. Como será notada, a própria denominação de cada camada acaba por contemplar sua funcionalidade:

- Camadas de dados: é onde estará localizado o arquivo físico de banco de dados, gerido por seu respectivo SGBD;
- Camada intermediária (camada servidora): camada que contempla o Servidor de Aplicação que, neste cenário, nada mais é do que uma aplicação Delphi tradicional, tal qual Desktop (VCL Forms, Windows Service ou Console Application) ou WebBroker;
- Camada cliente: engloba as eventuais aplicações clientes que irão acessar o Servidor de Aplicação. Por conseguinte, estas podem ser

dos mais variados tipos, incluindo as de âmbito nativo (VCL Forms Application, FireMonkey HD Application, FireMonkey Mobile Application, só para citar três) ou mesmo de outras tecnologias/linguagens, tais como Java, C#, PHP, entre outras.

Abordagem de exemplo

A fim de dar vida à contextualização exposta até aqui e formalizar concretamente a intenção do artigo, será usado como cenário base uma situação real de desenvolvimento multicamadas DataSnap tradicional, a partir do uso do dbExpress para toda a parte de conectividade (acesso a dados e cliente) existente. Por conseguinte, será mostrada sua eventual substituição pelo FireDAC. Dito isto, a seguir são elencados os principais pontos que envolvem a atividade como um todo.

Banco de Dados

Conforme mencionado, no Delphi, a construção de projetos DataSnap está intimamente ligada ao provimento de aplicações que envolvem dados, mais especificamente fontes de dados. Na prática, tais fontes nada mais são do que bancos de dados, relacionados aos mais variados tipos de SGBDs (Sistema de Gerenciamento de Banco de Dados), tais como Oracle, Firebird, InterBase, MySQL, SQL Server, dentre tantos outros. Logo, a escolha por determinada opção será fundamentalmente baseada no cenário a ser atendido, aliada à disponibilidade do contexto. Sendo assim, como exemplo natural, bancos tidos como sendo de "grande porte", tais como Oracle e SQL Server, são comumente os selecionados a atender aplicações de grandes corporações, enquanto que opções mais simplistas, vide Firebird e MySQL, acabam por satisfazer as necessidades de pequenos aplicativos domésticos/comerciais. Mediante isso, para uma exemplificação conceitual, duas opções melhor se enquadram – InterBase e Firebird – devido ao fato da praticidade provida por ambos. A primeira delas está ligada à distribuição e disponibilidade, uma vez que ambos podem ser obtidos de forma gratuita, sendo o InterBase também parte integrante da instalação padrão do próprio Delphi. Sem deixar de mencionar, a versão atual do IDE contempla a versão XE3 deste SGBD. Outro fator colaborativo se dá em razão da disponibilidade, em ambos os produtos, de um arquivo de banco de dados de exemplo, denominado EMPLOYEE (.FDB no Firebird e .GDB no InterBase), que faz com que nenhum tempo seja gasto com a construção de uma estrutura de dados ilustrativa. A seguir, são mostrados os diretórios padrão para o referido arquivo em suas respectivas distribuições:

Delphi XE6

```
C:\Users\Public\Documents\Embarcadero\Studio\14.0\Samples\Data\EMPLOYEE.GDB
```

Firebird 2.5

```
C:\Program Files (x86)\Firebird\Firebird_2_5\examples\empbuild\EMPLOYEE.FDB
```

Vale lembrar que toda esta proximidade entre tecnologias, Firebird e InteBase, se dá em razão do primeiro ter sido originado a partir do segundo, fato este que vai além do escopo do presente artigo.

Conexão com o banco

Diante das recentes versões do IDE, a construção de aplicações de banco de dados no Delphi parte da utilização do Data Explorer, recurso este que provê a estabilização e posterior manipulação de conexões ativas a fontes de dados, dentro do próprio ambiente de desenvolvimento. Assim, num primeiro momento, o desenvolvedor se vê livre da necessidade de instalação e utilização de uma ferramenta direcionada à parte, o que acaba por agilizar todo o trabalho. Até o Delphi XE5, o Data Explorer era um artefato voltado exclusivamente ao dbExpress, possibilitando o registro de conexões a toda sua gama de fontes de dados suportadas, incluindo SGBDs e servidores DataSnap. Todavia, com o advento do FireDAC, a versão atual do IDE traz agora um Data Explorer remodelado, pronto a atender as duas principais tecnologias nativas de acesso a dados atual, tal como mostra a **Figura 2**.

Mediante o exemplo proposto, do ponto de vista "tradicional", a definição de uma nova conexão dbExpress para Firebird (ou InterBase), direcionada ao banco de dados EMPLOYEE, já seria o suficiente para a fundação de dados da aplicação servidora DataSnap ou mesmo para uma aplicação Client/Server.

Servidor de Aplicação

Conforme já mencionado, no âmbito do Delphi, o conceito de multicamadas está atrelado ao uso da tecnologia DataSnap. Logo, do ponto de vista nativo, o provimento de um projeto deste tipo se dá inicialmente pela construção da aplicação servidora (Servidor de Aplicação) que, em tese, estará situada num servidor dedicado, uma vez que concentra todas as tratativas de acesso a dados e regras de negócio da conjuntura da aplicação, considerados dois dos pontos mais onerosos da mesma. Na prática, a construção de um Servidor de Aplicação pode ser feita basicamente de duas formas. A primeira, mais intuitiva, se dá pela utilização de assistentes inteligentes (Wizards) disponibilizados pelo próprio IDE, por meio de tipos de projetos específicos. Conforme pode ser visto na **Figura 3**, atualmente são três as opções de uso:

- **DataSnap REST Application**: opção que pode ser considerada a mais atual, dentre todas as demais disponíveis, com relação ao cenário atual do mundo do desenvolvimento de software. Isto porque, ela dá acesso a um assistente que servirá como ponto de partida à construção de uma aplicação servidora DataSnap habilitada para a Web, através do uso dos protocolos de comunicação HTTP e HTTPS, aliado ao envolvimento de tecnologias condizentes e plenamente utilizadas no âmbito corrente, tais como AJAX, JavaScript, JSON e REST (**BOX 1**);
- **DataSnap Server**: esta opção é considerada a mais tradicional, provendo um assistente que irá guiar o desenvolvedor na implementação inicial de uma aplicação servidora DataSnap, com suporte a todos os protocolos suportados atualmente pela tecnologia – TCP/IP, HTTP e HTTPS. Adicionalmente, tal aplicação pode ainda assumir três diferentes tipos – VCL Forms Application, Console Application ou Service Application (Serviço);
- **DataSnap WebBroker Application**: opção que provê, a partir de um Wizard específico, a implementação inicial de uma aplicação

servidora baseada essencialmente em duas tecnologias singulares – DataSnap e WebBroker. Esta última, em sua essência, acaba tornar o contexto com características do cenário Web, envolvendo a internet e o provimento de conteúdo aos clientes em formatos tradicionais deste segmento, tais como HTML e XML.

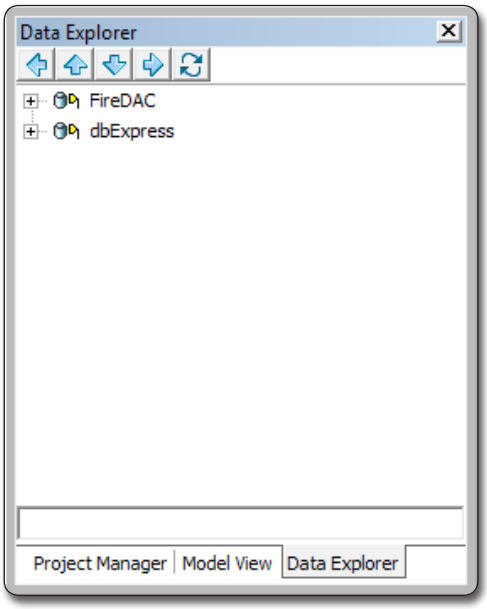

Figura 2. Delphi XE6 – Novo Data Explorer

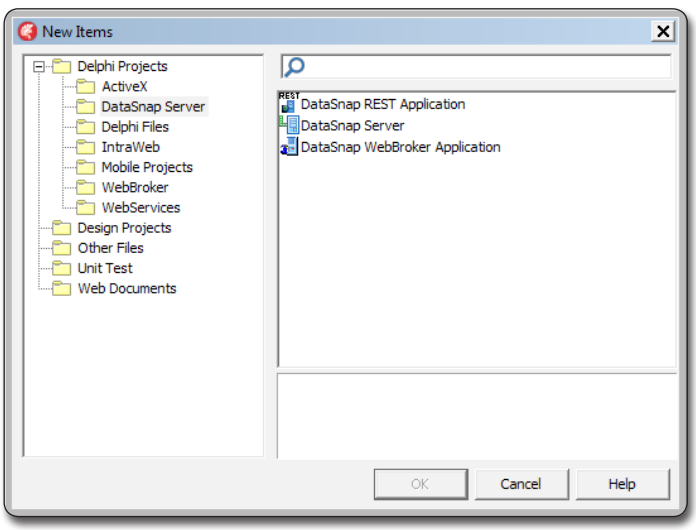

Figura 3. Tipos de Projetos DataSnap Server

Mediante os itens disponíveis, o cenário de um simples projeto de exemplo acaba por incidir naturalmente nas opções tidas como "tradicionais" que, neste caso, se tratam do item DataSnap Server em conjunto do tipo de projeto VCL Forms Application. Assim, ao seu utilizar tais opções, uma estrutura padrão é criada para o projeto, conforme pode ser visto na **Figura 4**. A seguir, para fins de conhecimento, é feita uma breve explanação de cada elemento presente:

• Unit1.pas: em se tratando de um projeto VCL Forms, tal arquivo representa o formulário principal da aplicação. Conforme já mencionado, uma aplicação servidora DataSnap deste tipo torna-se o ideal para o desenvolvimento inicial, uma vez que facilita todo o processo por meio de sua interface visual;

• ServerContainerUnit1.pas: este arquivo, conforme seu nome padrão sugere nada mais é do que o container principal do projeto, relacionado ao servidor. Logo, sua função principal é exclusivamente manter os componentes DataSnap que se fazem exclusivos a este cenário, tais como TDSServer, TDSTCPServerTransport e TDSServerClass. A fim de se preservar as tradições do Delphi, neste cenário, tal estrutura se reflete em um TDataModule;

• ServerMethodsUnit1.pas: no contexto DataSnap, existirão no lado Server 1 ou N estruturas que, fundamentalmente, conterão os métodos e elementos do servidor que serão expostos aos eventuais clientes que irão surgir. Por padrão, tais estruturas são personificadas em ServerMethods que, na prática, podem assumir vários tipos, incluindo containers (TDataModule ou TDSServerModule) ou somente código (TComponent). Assim, no caso de tratativas com banco de dados, o habitual é concentrar os componentes relacionados em um TDSServerModule, que já dispõe de todos os recursos necessários para a exposição de DataSets e Providers do lado servidor para o lado cliente nativo;

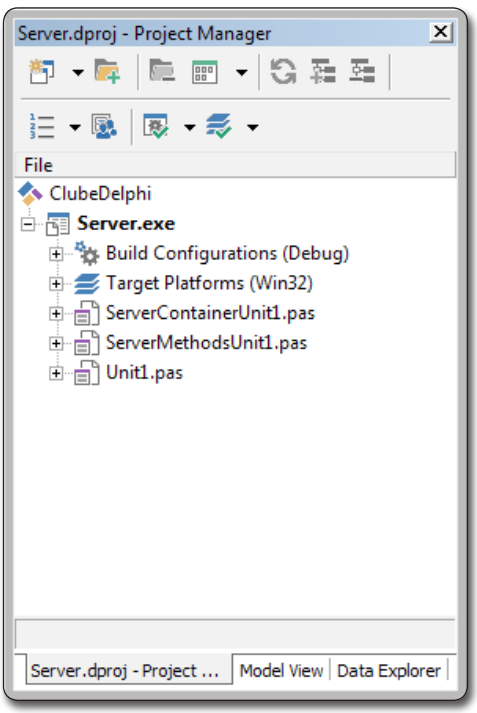

Figura 4. DataSnap Server – estrutura inicial do projeto

Em vista do que foi mencionado, numa exemplificação tida como "padrão", a centralização dos componentes de acesso a dados na unit ServerMethods é ilustrada na **Figura 5**. Tomando como base o IDE do Delphi XE6, tudo isso é conseguido de uma maneira muito prática através do uso da conexão previamente registrada no Data

Explorer, através de um simples arrastar-e-soltar do nó relativo à tabela desejada ao container utilizado. No caso da utilização do dbExpress, imediatamente após esta ação, são adicionados ao projeto um elemento TSQLDataSet direcionado à referida tabela, bem como um TSQLConnection (caso não haja um), devidamente configurado à conexão utilizada. Em complemento, visando a utilização deste DataSet numa camada cliente nativa (aplicação cliente Delphi) por meio de um TClientDataSet, adiciona-se um TDataSetProvider correlacionado ao TSQLDataSet.

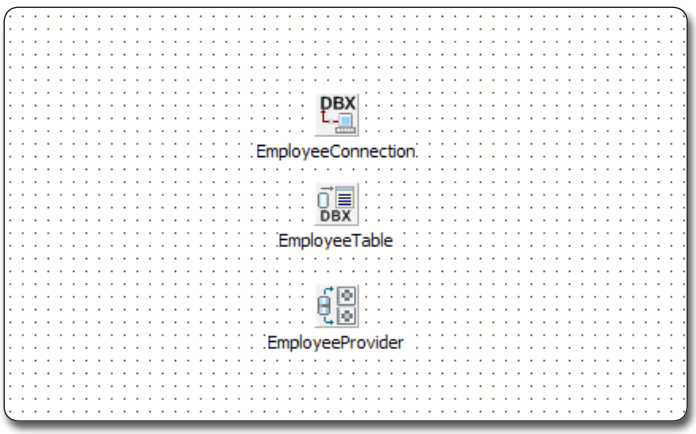

Figura 5. Container Servidor – componentes de acesso a dados dbExpress

BOX 1. JSON e REST

Java Script Object Notation, ou simplesmente JSON (lê-se djeisón), é um formato computacional que visa a troca de dados operacionais entre aplicações heterogêneas, sendo uma alternativa ao já tradicional XML. Neste âmbito, a simplicidade de sua sintaxe pode ser pontuada como sendo uma de suas principais características, visto que, tal fator faz com que uma enorme gama de distintas linguagens de programação o suportem atualmente, tais como: Delphi, C++, C#, Java, PHP, Python e Ruby, entre outras.

REST é atualmente a sigla utilizada no cenário de desenvolvimento de software para se referir ao termo em inglês Representational State Transfer que, numa tradução livre para o português seria algo como Transferência de Estado Representacional. Logo, de forma conceitual, REST pode ser definido como sendo um estilo de projetar aplicativos Web fracamente acoplados, cujos recursos são nomeados e acessíveis diretamente através de uma URI (Identificador Uniforme de Recursos) exclusiva para cada recurso.

Conexão com o Servidor

Partindo novamente do "tradicional", uma abordagem inicial bastante comum se dá pela construção de uma aplicação cliente do tipo VCL Forms. No IDE XE6, o primeiro passo para tal se estabelece por meio de uma boa prática, que ocorre antes mesmo do início do desenvolvimento efetivo, através da realização de um teste de conectividade com o Servidor de Aplicação existente. Apenas para retomar conceitos, vale lembrar que, nativamente, a conexão estabelecida entre as partes Server e Client se dá por meio de um driver dbExpress exclusivo, intuitivamente nomeado como DataSnap. Sendo assim, o Data Explorer entra novamente em cena, possibilitando o registro de uma conexão deste tipo, tanto para fins de testes de conectividade, quanto para uso posterior no projeto. Na prática, todo processo é bastante similar ao realizado para banco de dados, tendo início na definição de um nome para a conexão seguida pela posterior configuração de seus parâmetros (**Figura 6**).

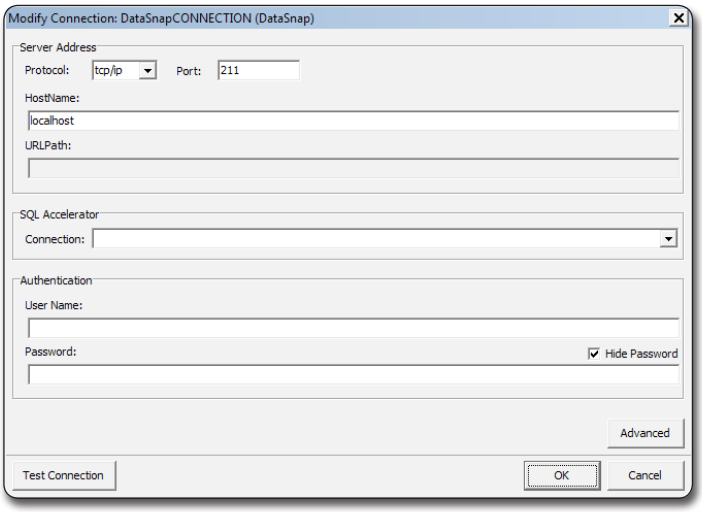

Figura 6. Data Explorer – Parâmetros da conexão DataSnap

Comumente, para um desenvolvimento introdutório local, as próprias atribuições padrão direcionadas aos parâmetros Protocol, Port e HostName são o suficiente para a estabilização da conexão desejada. Adicionalmente, neste instante, além do teste de conectividade, há também a possibilidade de provação do acesso aos métodos do servidor. Visando esta pontualidade, o próprio Wizard DataSnap Server provê, dentre suas opções, a concepção inicial de dois métodos simples de amostra (Sample Methods) na classe de Server Methods, denominados EchoString e ReverseString, que irão promover a testabilidade desejada. A seguir, são exibidas as declarações de ambos.

```
function EchoString(Value: string): string;
function ReverseString(Value: string): string;
```

Na estrutura do Data Explorer, os métodos remotos de um Servidor DataSnap ficam dispostos sob o nó Server Methods, e a chamada a cada um deles ocorre de maneira semelhante a uma chamada a Stored Procedure de um banco de dados, dada a manipulação de parâmetros de entrada e de saída, quando houverem. Os métodos na linguagem Delphi trazem consigo uma nomeação de parâmetros de entrada, o que não é válido para sua saída (retorno). Dado este fato, o Data Explorer estabelece como regra geral para este contexto o uso do nome "ReturnParameter" para todo retorno de método. Por fim, toda a parte de conectividade está pronta a operar em uma construção nativa.

Camada Cliente

A difusão e natural adoção da tecnologia DataSnap no desenvolvimento de aplicações Delphi de banco de dados se deve muito

ao fato de que, uma vez construído um Servidor de Aplicação, inúmeros são os tipos de clientes aptos a consumirem seus recursos. Somente citando o aspecto nativo, ou seja, dos tipos providos internamente por meio do IDE do Delphi, estão aplicações VCL Forms, FireMonkey Desktop e FireMonkey Mobile, que acabam por atender plataformas distintas, tais como Windows, Mac OS, iOS e Android, bem como dispositivos heterogêneos (PCs, Tablets e Smartphones).

Seguindo pela linha do "básico", a construção de uma aplicação cliente nativa tradicional é refletida no tipo de projeto VCL Forms Application. De forma natural, esta segmentação acaba por se alinhar de forma bastante positiva e facilitada ao contexto existente, uma vez que está apta a tirar proveito de um emaranhado de proventos proporcionados por diversos componentes da extensa biblioteca nativa do Delphi. Para este caso, num primeiro plano, é possível citar a conectividade cliente provida pelo dbExpress, por meio de seu driver exclusivo, bem como pelo uso de objetos TClientDataSet, que estarão "atrelados" aos Providers remotos.

Na prática, uma vez que se tem definida, no Data Explorer, uma conexão ativa ao Servidor de Aplicação, o simples arrastar e soltar desta em direção a um elemento container do projeto, é o suficiente para que um componente de conexão dbExpress seja adicionado a ele, já totalmente configurado à mesma, tal como ocorre com conexões a banco de dados.

Migração de dbExpress para FireDAC

Mediante o conceito proposto pelo fundamento das aplicações DataSnap, cada parte de sua arquitetura deve ser substituível, dando a flexibilidade necessária para que uma eventual troca de uma de suas partes não impacte de forma negativa nas demais ou mesmo no projeto como um todo. Neste sentido, alguns exemplos habituais tidos como "clássicos" podem ser citados, tais como:

• Troca de tecnologia de banco de dados (SGBD): no contexto DataSnap, o banco de dados, de forma conceitual, é interpretado como um mero repositório de informações. Assim, nenhuma regra de negócio estará definida ou presente em sua estrutura, tal como no caso de Stored Procedures funcionais. Tudo isso visa atingir um nível ideal de independência de tecnologia de banco de dados, fazendo com que uma eventual troca de SGBD ocorra de maneira transparente e sem impactos, ao âmbito da aplicação;

• Troca de tecnologia cliente: neste cenário, não há uma troca efetiva de tecnologia cliente, mas sim a aceitação a uma diversidade de opções distintas. Sendo assim, partindo de um Servidor DataSnap idealmente construído, a característica da aplicação cliente torna-se indiferente. Isso acaba por incluir tanto construções nativas quanto aquelas providas por outras tecnologias, tais como Java, .NET, PHP, entre outras;

• Troca de tecnologia de acesso a dados: principalmente em cenários críticos que envolvem dados, não é fantasiosa a necessidade de troca/substituição, seja por qual for o motivo, da tecnologia de acesso a dados utilizada até então, por outra que venha a atender aos anseios arquiteturais, de negócio ou mesmo comerciais do momento. Logo, uma construção multicamadas DataSnap tende a facilitar este tipo de ação, uma vez que os componentes relacionados se concentram em apenas uma camada da arquitetura. Além disso, neste tipo de abordagem, os elementos de acesso a dados não estão relacionados diretamente a nenhuma perspectiva do cliente.

Em vista de tudo isso, dois pontos podem e devem ser ressaltados neste momento de forma a estruturar aspectos sutis, porém determinantes, no desenvolvimento multicamadas com DataSnap no Delphi. Ambos estão relacionados à migração "dbExpress to FireDAC", que tende a ser uma das vertentes mais exploradas em meio à comunidade, uma vez que recentemente o segundo (FireDAC) foi posto, pela própria fabricante (Embarcadero), como sendo o sucessor natural e ideal do primeiro (dbExpress). O primeiro aspecto se fundamenta no fato de que o dbExpress serviu como "base" para a arquitetura interna do "novo DataSnap", apelido este surgido ainda na época do Delphi 2009, no momento da total reestruturação do framework. Adicionalmente, isso se reflete de forma direta no lado prático, dada a facilidade e intuição em se direcionar a um elemento da biblioteca DBX (TSQLConnection), a parte de conectividade do cliente com o servidor. Já o segundo aspecto, também atrelado a este ponto, diz respeito ao driver nativo dbExpress, exclusivo à tecnologia DataSnap que, devida sua excelência operacional, torna-se um item valioso no contexto do desenvolvimento. Por conseguinte, a opção por sua utilização acaba sendo algo mecânico, em vista de todas as literaturas surgidas desde o lançamento do "novo DataSnap", ser baseada em seu uso.

Por tudo isso, no momento atual, é possível afirmar que é improvável uma total dissociação do uso dbExpress em projetos DataSnap. Todavia, em função de toda a mitificação dada ao FireDAC, é possível presumir, para um futuro próximo, a total estabilização desta tecnologia em todos os aspectos aqui mencionados. Obviamente, o desenvolvedor mais coerente levará em conta, em caso de uma possível migração, todos os aspectos positivos e negativos que permeiam tal realização, evitando modismos e ressaltando a sobriedade.

Lado Servidor

Todo o reflexo proporcionado pela migração citada incide de início, quase que exclusivamente sobre a camada Server, onde está localizada o Servidor de Aplicação DataSnap propriamente dito. Isso se deve ao fato de que é nesta estrutura onde estão centralizadas as tratativas de banco de dados, por meio dos componentes de acesso a dados pertencentes à biblioteca utilizada, neste caso o dbExpress. Assim, em termos gerais, deverá haver uma troca física de tais elementos por outros correlacionados, porém agora pertencentes à biblioteca do FireDAC.

Mediante o cenário ilustrativo apresentado anteriormente, sua nova formatação, nesta situação, se daria de forma semelhante à mostrada na imagem a seguir (**Figura 7**). Logo, de acordo com o apresentado, os seguintes objetos foram incorporados, em substituição aos tradicionais elementos DBX:

• TFDConnection (EmployeeConnection): elemento FireDAC que substitui o TSQLConnection. Dessa forma, refere-se ao componente responsável pelo estabelecimento da conexão com o SGBD. Nos bastidores, sua classe apresenta-se sob o nome de TFDConnection, descendente do tipo TFDCustomConnection, e que adicionalmente, acaba por contemplar uma série de recursos que irão promover o gerenciamento de todos os DataSets associados à conexão;
• TFDQuery/TFDTable (EmployeeTable): no popular, tais elementos são brevemente interpretados como sendo "DataSets" do FireDAC. Assim, as classes de ambos implementam aditivos responsáveis por prover uma "ligação" a um conjunto de dados provindos do banco de dados. Na prática, no caso do TFDQuery, isso se dá por meio da execução de uma consulta SQL, enquanto que para o TFDTable há um apontamento direto a uma única tabela do banco. Aqui, o uso de um ou outro ocorre em substituição ao DataSet do dbExpress (neste caso, TSQLDataSet). Em termos de característica, os DataSets FireDAC se diferem dos DataSets DBX pelo fato de contemplarem internamente, recursos de edição e navegação bidirecional de dados. Isso implicaria, por exemplo, na não necessidade de utilização complementar da dupla TDataSetProvider+TClientDataSet em uma abordagem Cliente/Servidor;
• TFDGUIxWaitCursor: componente de uso exclusivo da segmentação FireDAC, cuja funcionalidade se dá pelo controle de um cursor de espera padrão. Pelo lado técnico, faz com que ocorra a implementação da interface IFDGUIxWaitCursor na aplicação, que é item obrigatório em contextos desta tecnologia. Dada sua importância nos bastidores, apenas a declaração de suas units relacionadas na seção uses do projeto se fazem necessárias.

Lado Cliente

Sob a perspectiva nativa RAD, de construções exclusivamente Delphi, a busca por uma ambientação inteiramente FireDAC também no lado cliente, se reflete em duas alternativas práticas iniciais. A primeira está relacionada ao consumo de métodos remotos "simples", cujo retorno é baseado em tipos primitivos conhecidos, vide String, Integer, Double, etc. Como exemplo para esta abordagem, é possível citar os próprios métodos declarados previamente no Servidor DataSnap – EchoString e ReverseString – que retornam um valor String. Já neste momento vale ressaltar que, independente da abordagem utilizada, assim como ocorre com o dbExpress, um padrão de uso tende a se estabelecer. Para o caso de aplicações clientes VCL Forms, quatro componentes surgem como essenciais para este cenário (**Figura 8**), cuja maioria já faz parte do uso tradicional da biblioteca:
• TFDConnection (ClientConnection): componente FireDAC que trata de toda a parte de conectividade de uma aplicação. Neste cenário, age como o elemento responsável por estabelecer a conexão da parte Cliente com o Servidor;
• TFDStoredProc (TFDStoredProc1): componente voltado à execução de procedimentos armazenados no lado Servidor, o que inclui objetos de banco de dados (Stored Procedures) bem como métodos remotos. Este último, seu foco na atual conjuntura;
• TFDPhysDSDriverLink (TFDPhysDSDriverLink1): objeto que vincula o driver FireDAC DataSnap a aplicação. Na prática, pode seu uso pode ser suprimido pela simples declaração de sua unit relacionada (FireDAC.Phys.DS) na seção uses da aplicação;
• TFDGUIxWaitCursor (TFDGUIxWaitCursor1): elemento tido como "necessário" ao âmbito do desenvolvimento FireDAC, conforme já explicitado anteriormente.

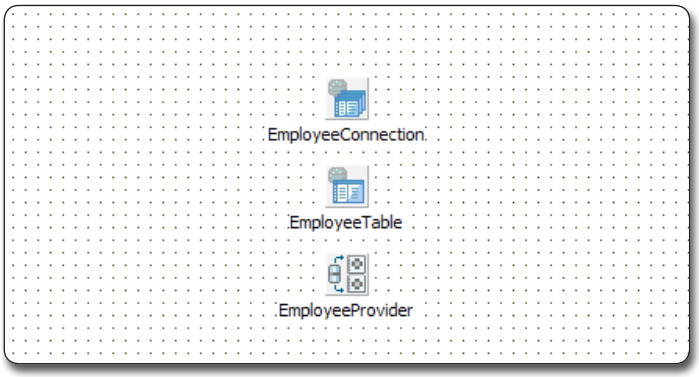

Figura 7. Container Servidor – componentes de acesso a dados FireDAC

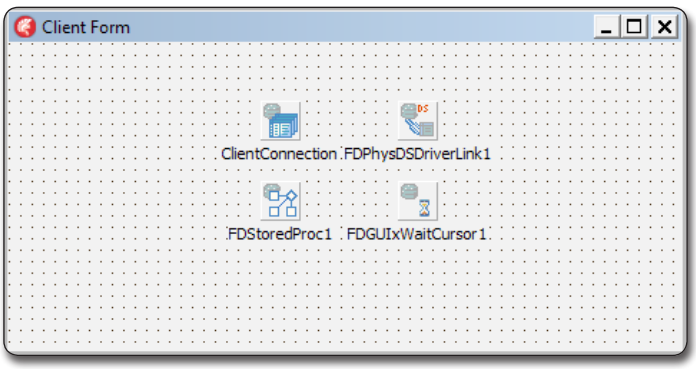

Figura 8. Aplicação Cliente – componentes FireDAC

Consumindo métodos do Servidor

A configuração básica para esta ambientação FireDAC no lado cliente ocorre fundamentalmente em dois dos elementos apresentados. Por questões óbvias, a primeira se dá sobre o componente de conexão (TFDConnection), num processo muito semelhante ao realizado em abordagens com o dbExpress, onde basicamente somente a informação do Driver é o suficiente (subentendo o uso da porta padrão 211). Tudo isso é ilustrado a seguir através de um trecho de seu código DFM. Sem deixar de citar, até como forma de justificar a já citada e notória dependência atual do DataSnap para com o dbExpress, aqui o driver DataSnap do FireDAC se do próprio driver DataSnap do dbExpress, através uma linkagem direta estática.

```
object FDConnection1: TFDConnection
  Params.Strings = ('DriverID=DS')
  ...
end
```

Uma vez conectado ao Servidor DataSnap, a aplicação cliente está apta a operar com seus métodos disponíveis. Conforme citado, isso irá ocorrer em torno de um elemento do tipo TFDStoredProc que será, portanto, uma representação física no lado cliente, de um determinado método remoto. Tomando como base a função ReverseString do servidor, a primeira configuração a ser feita no componente ocorre em sua propriedade Connection, que irá receber o elemento de conexão (ClientConnection) ao qual estará vinculado. Ao fazer isso, imediatamente é carregada em sua propriedade SchemaName, uma lista contendo os contextos de operações disponíveis (**Figura 9**), incluindo a classe remota TServerMethods1, onde está declarado o método desejado. Uma vez selecionado o "schema", seus recursos (métodos) associados são disponibilizados na propriedade StoredProcName, tal como pode ser visto na **Figura 10**. Por fim, em se tratando de código, a chamada do método acontece de maneira similar ao uso com objetos de banco de dados do gênero, conforme mostra a **Listagem 1**. Como resultado, a **Figura 11** ilustra a mensagem obtida pela execução do trecho mostrado.

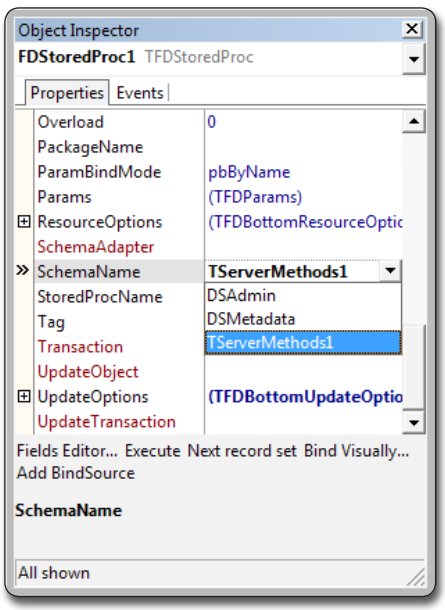

Figura 9. TFDStoredProc – propriedade SchemaName

Listagem 1. Código de exemplo para chamada ao método remoto por meio do componente TFDStoredProc

```
1.  ...
2.  FDStoredProc1.Close;
3.  FDStoredProc1.Params[0].AsString := 'CLUBE DELPHI';
4.  FDStoredProc1.ExecProc;
5.  ShowMessage(FDStoredProc1.Params[1].AsString);
6.  ...
```

Consumindo "DataSets" do Servidor

Outra alternativa prática neste cenário se dá pelo consumo de DataSets remotos no lado cliente. Novamente pautando pelo uso de construções nativas, o uso dbExpress se refletia no uso dos seguintes componentes:

- TSQLConnection: objeto de conexão que aponta para o Servidor de Aplicação;
- TDSProviderConnection: objeto que representa um "schema" remoto, mais precisamente uma classe exposta do servidor, tal como a referida TServerMethods1;
- TClientDataSet: elemento que estará ligado a um DataSetProvider remoto por meio de um ProviderConnection.

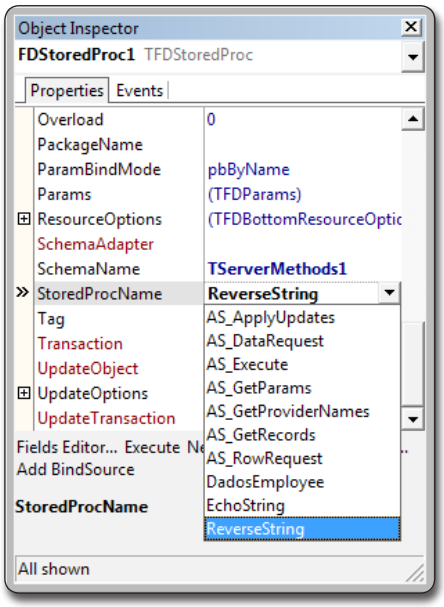

Figura 10. TFDStoredProc – propriedade StoredProcName

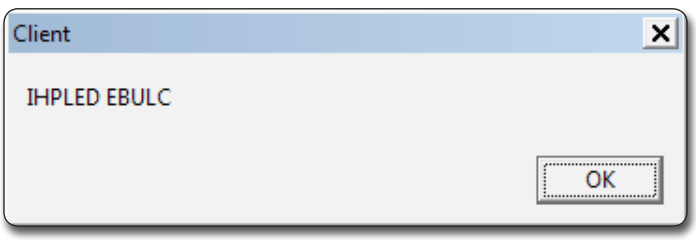

Figura 11. Aplicação Cliente – chamada ao método Reverse String

Sob essa nova perspectiva proposta, todo este panorama é alterado, com a substituição desses elementos pelos componentes FireDAC tidos como "essenciais" numa construção cliente: TFDConnection, TFDStoredProc, TFDPhysDSDriverLink e TFDGUIxWaitCursor. Atrelado a isso, está a inclusão de um novo método remoto na classe servidora (TClubeDelphiServerMethods), que irá retornar um objeto do tipo TDataSet, conforme a seguir:

```
function TServerMethods1.DadosEmployee: TDataSet;
begin
  EmployeeTable.Open;
  Result := EmployeeTable;
end;
```

Mediante isso, tal como se deu na abordagem anterior, este novo método poderá ser recuperado no lado cliente através de um novo elemento TFDStoredProc. Por conseguinte, o parâmetro existente é automaticamente mapeado ao componente, tal qual um ResultSet, o que permite seu uso como um DataSet local. Tudo isso pode ser conferido no código DFM mostrado a seguir, na **Listagem 2**.

Listagem 2. Objeto FDStoredProc2

```
1. object FDStoredProc2: TFDStoredProc
2.   Connection = FDConnection1
3.   SchemaName = 'TServerMethods1'
4.   StoredProcName = 'DadosEmployee'
5.   ParamData = <
6.    item
7.      Name = 'ReturnValue'
8.      DataType = ftDataSet
9.      ParamType = ptResult
10.     DataTypeName = 'TDBXTypes.TABLE'
11.   end>
12.end
```

Logo, o uso de um TDataSource ligado ao elemento TFDStoredProc bem como a um controle visual TDBGrid, é o suficiente para possibilitar a visualização em tela dos dados condizentes ao retorno do método remoto (**Figura 12**). Em runtime, isso é conseguido através de uma simples chamada ao método Open do componente.

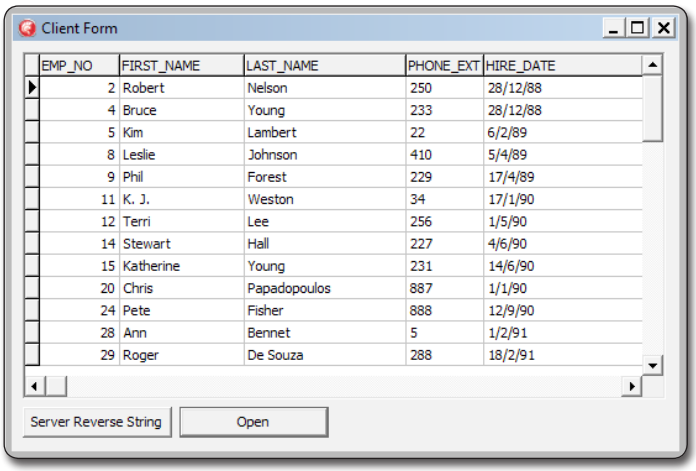

Figura 12. Aplicação Cliente – visualização de dados do Servidor

Assim, de agora em diante, para a construção de novas aplicações de banco de dados, incluindo multicamadas, o desenvolvedor Delphi passa a ter no FireDAC, uma de suas principais alternativas no que diz respeito à parte de acesso a dados do projeto. Atrelado a este cenário estão também as aplicações legadas que, num processo de migração, tal como o mostrado neste artigo, podem tirar proveito desta nova tendência que se estabelece no âmbito da ferramenta.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br

Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

RAD Studio XE6 – Página oficial do produto
http://www.embarcadero.com/products/rad-studio

RAD Studio XE6 – Download Trial
https://downloads.embarcadero.com/free/rad_studio

Firebird – Página oficial do projeto
http://www.firebirdsql.org

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

SEU CARRO TEM SEGURO, SUA SAÚDE TEM SEGURO, MAS E O SEU EMPREGO... TÁ SEGURO??

NÃO DEIXE JUSTAMENTE A SUA CARREIRA FICAR EM RISCO!

Manter-se atualizado com todas as novidades do mercado de desenvolvimento é obrigação de todo bom programador. Faça agora mesmo um seguro para a sua carreira. Seja um assinante MVP!

Saia do risco!

TENHA ACESSO A:

- +DE 290 CURSOS ONLINE
- 09 REVISTAS MENSAIS
- 9.074 VÍDEO-AULAS

POR APENAS 69,90 MENSAIS

ASSINATURA MVP MOST VALUABLE PROFESSIONAL

QUEM TEM ESTÁ TRANQUILO.

DEVMEDIA | Acesse: www.devmedia.com.br/mvp

EntityDAC: Ferramenta de Mapeamento Objeto-Relacional – Parte Final

Este artigo faz parte de um curso

Fique por Dentro

O presente artigo dá sequência à abordagem que trata do EntityDAC, produto recém-lançado pela empresa Devart, que atua como uma ferramenta de mapeamento objeto-relacional voltada exclusivamente ao contexto Delphi. Logo, sua intenção fundamental é a de prover um mecanismo facilitador no que diz respeito à construção de aplicações de banco de dados, norteadas pela essência do paradigma da POO (Programação Orientada a Objetos), mais especificamente no tratamento da relação a ser estabelecida entre os objetos da aplicação e as tabelas de bancos de dados relacionais. Neste sentido, o aprendizado prático do EntityDAC se torna útil inicialmente pela plena visualização dos pontos que diferem uma abordagem deste tipo em relação a tradicional e, posteriormente, na colaboração para sua utilização no cotidiano do desenvolvedor Delphi.

De forma simples, o EntityDAC pode ser definido como um framework ORM para Delphi. Esta sigla faz referência ao termo em inglês Object-Relational Mapping que, numa tradução livre para o português, seria algo como "Mapeamento Objeto-Relacional". No desenvolvimento de software, este termo caracteriza uma técnica que está associada ao uso da Programação Orientada a Objetos (POO) no provimento de aplicações de banco de dados. Neste cenário, o contraponto existente numa construção OO, em relação a uma construção tradicional, se dá essencialmente pelo envolvimento de Entidades (classes) e suas instâncias (objetos) na aplicação, ao invés de dados simples provindos de determinada fonte de dados. Esse fator dá origem a uma impedância natural existente entre o contexto de objetos da aplicação com o uso de um banco de dados relacional, que somente opera dados singulares. Em outras palavras, a instância de uma entidade possui uma estrutura complexa o suficiente para conter desde atributos simples a entidades aninhadas, o que impede a persistência direta dessas informações numa tabela do banco de dados, sendo necessária uma tratativa pontual. Logo, o mapeamento objeto-relacional vem com o intuito de sanar este impedimento, mapeando os elementos de banco de dados, tal como Tabelas e Views, para classes de negócio que estarão presentes no código da aplicação.

No geral, o mapeamento realizado pelo EntityDAC dá origem a classes descendentes de um tipo específico, provenientes de seu próprio contexto. Tal tipo refere-se a classe básica do framework, denominada TEntity. Todavia, visando dar uma maior abertura e liberdade ao desenvolvedor, tal ancestral pode ser plenamente substituído pelo tipo TObject, nativo do Delphi. Neste caso, o resultado final obtido é um conjunto de classes tidas como puras, uma vez que não apresentam dependência direta com classes fora do escopo do Delphi. Com o provimento de ambos os cenários, a intenção do EntityDAC é tornar seu uso facilitado tanto em novas construções quanto em projetos já existentes.

Sob o aspecto do Delphi, o uso de classes remete ao uso de todos os recursos do paradigma da Orientação a Objetos, tais como herança, polimorfismo e encapsulamento. Além disso, o EntityDAC é tido como um framework pelo fato de prover um conjunto de classes e componentes bem definidos que interagem entre si por um propósito em comum que, neste caso, é o próprio mapeamento objeto-relacional.

Terminologia EntityDAC

Em se tratando de um produto de características peculiares, o EntityDAC carrega consigo um agrupamento de terminologias que estão associadas à sua área de atuação. Em função disso, para o início de seu estudo, o pleno conhecimento e entendimento de tais termos torna-se algo essencial para o bom aproveitamento de seus recursos na prática, assegurando uma utilização ideal por parte do desenvolvedor. A seguir são então pontuados alguns dos termos mais utilizados no cenário do framework.

• **Model (Modelo) -** Em suma, refere-se à representação de elementos e ambientes que estão inseridos ao âmbito de trabalho do EntityDAC.
• **Database Model (Modelo de Banco de Dados) -** Representação do modelo de dados relacional por meio do conjunto de elementos presentes no banco de dados, tais como suas tabelas (e relacionamentos), campos (Fields), visões (Views), etc.
• **Object Model (Modelo de Objeto) -** Representação do conjunto de classes Delphi relacionadas ao negócio. Na literatura do EntityDAC, é também tratado apenas como Modelo.
• **Meta-Model (Modelo de Metadados) -** Representa um conjunto de classes Delphi tidas como "especiais", que de uma forma própria, descrevem o Modelo de Banco de Dados. Dessa forma, nelas é onde estarão armazenadas as informações relativas aos elementos de banco de dados o que envolve, por exemplo, a descrição das tabelas e de seus campos. Assim, o framework se utiliza dessas informações para realizar uma correta interação com o próprio banco, tais como nos processos de recuperação e persistência de informações.
• **Entity (Entidade) -** De forma geral, Entity é um elemento projetado exclusivamente ao cenário do EntityDAC. Como definição, uma entidade nada mais é do que uma representação abstrata de um objeto do Modelo de Banco de Dados no Modelo de Objeto. No código, cada entidade dá origem a uma classe Delphi. Por se tratar de um elemento direcionado, uma entidade apresenta algumas peculiaridades que tornam seu uso mais conveniente ao desenvolvedor. A principal delas está relacionada ao seu ciclo de vida, que fica a cargo de um mecanismo próprio do framework, denominado Data Context.
• **Data Context (contexto de dados) -** É o mecanismo responsável por gerenciar cada entidade de uma aplicação EntityDAC, no que diz respeito ao seu ciclo de vida. Logo, suas principais funções envolvem: criação e inicialização de novas instâncias de entidades, recuperação de entidades a partir do banco de dados, persistência de entidades no banco de dados, armazenamento em cache de entidades para utilização recorrente e eliminação de entidades não mais utilizadas.
• **LINQ -** O EntityDAC oferece suporte a diversos SGBDs (Sistemas de Gerenciamento de Banco de Dados), logo, uma aplicação baseada em sua estrutura estará apta a trabalhar com bancos de dados distintos. Sob uma visão tradicional, isso acarretaria o surgimento de alguns aspectos impeditivos, sendo o principal deles a especificidade existente em termos de sintaxe SQL (Structured Query Language), relacionada a cada SGBD. Visando contornar de forma ideal e elegante este tipo de barreira, o EntityDAC foi projetado a operar com uma linguagem de consulta padrão própria, denominada LINQ, em substituição à SQL. Tida como única, seu uso está relacionado à gama completa de bancos suportados pelo framework. Em termos conceituais, LINQ é o acrônimo para o termo em inglês Language Integrated Query, ou Consulta Integrada à Linguagem, no português. Conforme seu nome sugere, na prática, ela se mostra uma linguagem a ser utilizada em meio ao próprio código Delphi. Sendo assim, diferente da SQL, que é tratada como texto simples, o uso do LINQ é refletido em código programático, o que vem a habilitar o uso natural do completador de código (Code Completion) do IDE, bem como ter sua sintaxe analisada pelo compilador no momento da compilação.

Entity Developer

Pode ser considerado o IDE de desenvolvimento do EntityDAC. Por meio dele é que são feitas todas as definições de infraestrutura da aplicação, pertinentes ao framework. Devido sua usabilidade, o Entity Developer se integra ao próprio ambiente do Delphi, ficando acessível por meio do menu EntityDAC, conforme mostra a **Figura 1**.

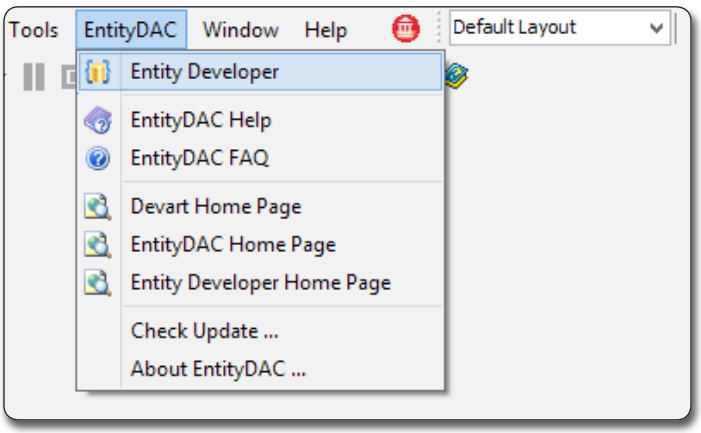

Figura 1. Entity Developer no IDE do Delphi

Ponto de partida com o EntityDAC

O ponto de partida para o uso do EntityDAC se dá pela escolha de uma, dentre três abordagens disponibilizadas pela ferramenta. Logo, cada abordagem caracteriza uma maneira diferente de iniciar o desenvolvimento, variando de forma fundamental no feitio com que será definido o Modelo e, consequentemente, as classes da aplicação. Nominalmente elas são apresentadas como Database-First, Model-First e Code-First.
• **Database-First -** A abordagem denominada Database-First caracteriza um desenvolvimento baseado em um banco de dados existente, o que justifica seu nome. Em vista disso, o mapeamento objeto-relacional realizado pelo EntityDAC toma como base a própria estrutura dos elementos detectados no banco para gerar as classes de negócio da aplicação. Logo, por se tratar de um processo todo automatizado, esta abordagem é tida como uma das mais usuais para o aprendizado e construções iniciais com o framework.
• **Model-First -** Model-First determina uma abordagem que, de acordo com sua nomeação, é pautada na elaboração do Modelo em primeiro lugar. Sendo assim, através de uma ferramenta exclusiva, disponibilizada pelo próprio EntityDAC, o desenvolvedor fará inicialmente a construção de todo o modelo da aplicação. Uma vez elaborada, tal estrutura dará então origem às classes de negócio bem como ao próprio banco de dados. Por este fato, aqui o desenvolvedor tende a centrar suas atenções exclusivamente nas

nuances do negócio, objetivando a determinação de um modelo sólido a perdurar por todo o ciclo de vida do projeto. Nestes casos, o banco de dados acaba por torna-se apenas um mero detalhe diante de todo o contexto.

• **Code-First -** Diferente das outras duas opções apresentadas, a abordagem Code-First prima pela construção inicial de código. Dessa forma, nenhum banco de dados ou modelo é exigido como ponto de partida, mas sim apenas a codificação das classes de negócio da aplicação. Por ter seu foco voltado ao código propriamente dito, esta abordagem tende ser muito apreciada e utilizada por desenvolvedores de nível mais avançado, que primam pelo completo domínio de todo o código presente no âmbito da aplicação.

Estudo de caso

Em meio ao desenvolvimento de software, o pleno aprendizado de uma nova abordagem ou tecnologia acaba por envolver duas fundamentações essenciais, resultantes na exposição conceitual e demonstração prática do assunto. Assim, uma vez apresentados e revistos os principais conceitos que estão relacionados ao EntityDAC, a sequência do estudo irá dar mostras dos pontos condizentes à utilização da ferramenta na construção de aplicações Delphi de banco de dados. Vale lembrar que a primeira parte desta série apresentou todo um embasamento teórico sobre o tema, bem como uma abordagem prática inicial, refletida na construção do Modelo (Object Model). Logo, a amostragem pautada a partir deste ponto tende a complementar o cenário previamente apresentado. Todavia, a intenção aqui não é a de prover um guia passo a passo, mas sim explicitar os pontos de conceitos a serem encontrados durante o desenvolvimento com o uso do framework.

Requisitos iniciais

Sem deixar de mencionar, a construção do modelo citado teve como base a abordagem Database-First do EntityDAC, a partir do banco de dados Employee, que acompanha a instalação do Firebird. Para o tópico, a versão do SGBD envolvida foi a 2.5. Naquele momento, por meio do IDE Entity Developer, foi feito uso do assistente Create Model Wizard para a criação efetiva do Modelo, cujo o arquivo de diagrama foi nomeado sugestivamente como ClubeDelphiModel.enml. Pelo fato de se tratar de um demonstrativo inicial, o modelo gerado acabou por englobar essencialmente todas as dez tabelas do referido banco. Visando dar uma maior vivacidade ao cenário e facilitar ainda mais o entendimento, alguns ajustes serão ilustrados. Uma vez que a forma com que o modelo pode ser concebido já é conhecida, o uso de um ou outro banco passa a ser mero detalhe. Logo, a **Listagem 1** mostra as instruções de definição de dados (DDL) referentes à criação de uma base customizada, que apresenta em sua estrutura uma única tabela, direcionada a um controle fictício de revistas. Em complemento, a **Listagem 2** exibe o script referente à carga inicial de dados para o contexto.

O Modelo

Já sob a perspectiva do IDE do Entity Developer, a **Figura 2** exibe o diagrama do modelo definido a partir do banco de dados criado, e que segue as mesmas diretrizes do modelo ilustrado no primeiro artigo desta série: Database-First para a abordagem e Code-mapped Entities para o tipo de mapeamento.

Listagem 1. Script DDL de criação da estrutura do banco de dados de exemplo

```
01.  SET SQL DIALECT 3;
02.  SET NAMES WIN1252;
03.  SET CLIENTLIB 'fbclient.dll';

04.  CREATE DATABASE 'C:\ClubeDelphi\DB\CLUBEDELPHI.fdb'
05.  USER 'SYSDBA' PASSWORD 'masterkey'
06.  PAGE_SIZE 8192
07.  DEFAULT CHARACTER SET WIN1252 COLLATION WIN1252;

08.  CREATE TABLE REVISTA (
09.    REVISTA_ID    INTEGER NOT NULL,
10.    NOME    VARCHAR(50),
11.    TITULO_CAPA    VARCHAR(50),
12.    NUMERO_EDICAO  INTEGER,
13.    QTD_PAGINAS    INTEGER
14.  );

15.  ALTER TABLE REVISTA ADD CONSTRAINT PK_REVISTA PRIMARY KEY
     (REVISTA_ID);
```

Listagem 2. Carga inicial de dados

```
01.  INSERT INTO REVISTA (REVISTA_ID, NOME, TITULO_CAPA, NUMERO_EDICAO,
     QTD_PAGINAS) VALUES (1,'CLUBE DELPHI','NOVIDADES DO DELPHI XE5', 155, 100);
02.  INSERT INTO REVISTA (REVISTA_ID, NOME, TITULO_CAPA, NUMERO_EDICAO,
     QTD_PAGINAS) VALUES (2,'CLUBE DELPHI','DBEXPRESS PARA FIREDAC', 153, 98);
03.  INSERT INTO REVISTA (REVISTA_ID, NOME, TITULO_CAPA, NUMERO_EDICAO,
     QTD_PAGINAS) VALUES (3,'CLUBE DELPHI','EXPLORANDO O DATASNAP', 151, 97);
04.  INSERT INTO REVISTA (REVISTA_ID, NOME, TITULO_CAPA, NUMERO_EDICAO,
     QTD_PAGINAS) VALUES (4,'CLUBE DELPHI','ANYDAC', 150, 90);
05.  INSERT INTO REVISTA (REVISTA_ID, NOME, TITULO_CAPA, NUMERO_EDICAO,
     QTD_PAGINAS) VALUES (5,'CLUBE DELPHI','MINERAÇÃO DE DADOS', 146, 80);
```

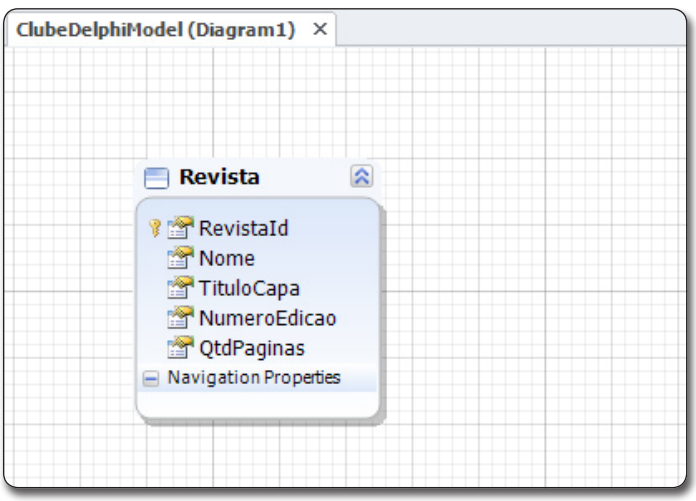

Figura 2. Diagrama do Modelo

Arquivos de código

A intenção principal deste demonstrativo é prover a visualização dos dados da tabela do banco de dados em uma aplicação Delphi, por meio do uso EntityDAC. Sendo assim, ao invés dos típicos DataSets para lidar com dados simples, neste tipo de abordagem

são utilizados elementos direcionados a trabalhar com entidades de negócio. A criação do diagrama do modelo acaba por definir visualmente um panorama das entidades envolvidas no desenvolvimento, porém, para a programação propriamente dita, o importante aqui é o código gerado nos bastidores, que dará origem aos arquivos de código-fonte que estarão presentes no projeto da aplicação. Isso é então conseguido através da opção Generate Code, presente no menu Model (**Figura 3**) do Entity Developer. Ao término do processo, uma janela informativa é apresentada, denominada Generation Process, que mostra o resultado da operação (sucesso ou falha). No caso de sucesso, a lista de arquivos gerados é exposta, conforme pode ser vista na **Figura 4**. Todos eles são automaticamente salvos no mesmo diretório onde está localizado o arquivo do diagrama do Modelo.

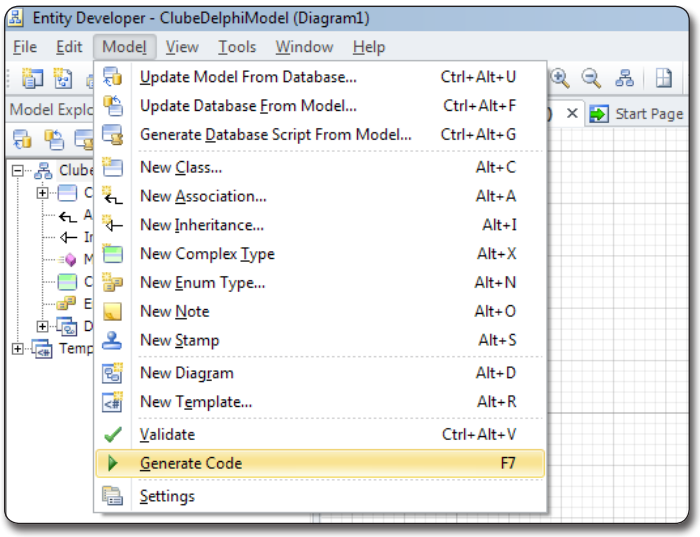

Figura 3. Opção Generate Code

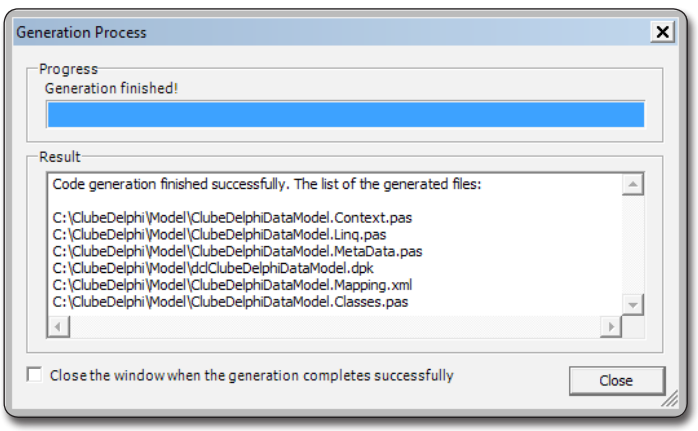

Figura 4. Generation Process – resultado da operação

A aplicação Delphi

De posse dos arquivos de código gerados pelo EntityDAC, o próximo passo natural é então adequá-los ao projeto de uma aplicação Delphi. Logo, a partir de um projeto VCL Forms Application tradicional, a inclusão destes arquivos se dá de forma padrão, numa simples adição de cada unit por meio do menu de contexto Add no Project Manager. Com base nisso, a **Figura 5** mostra a estrutura inicial citada.

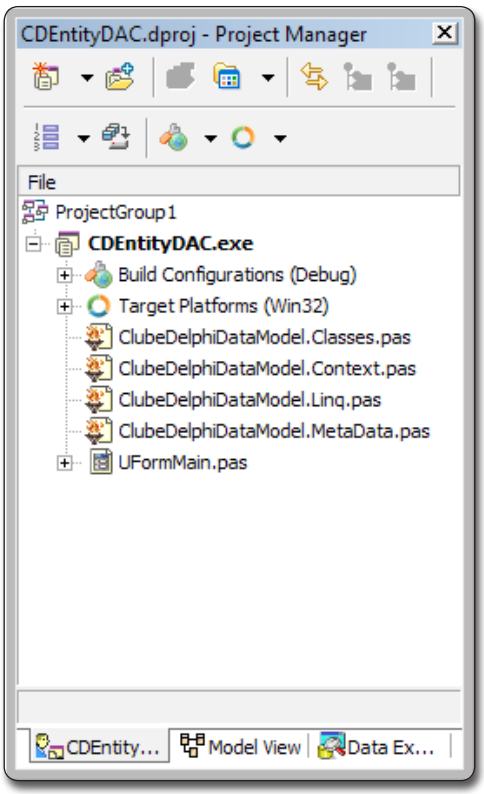

Figura 5. Project Manager – estrutura inicial do projeto

Conexão com o banco de dados (TEntityConnection)

Apesar de estar diretamente relacionado à construção de aplicações de banco de dados, o EntityDAC não reflete exatamente uma camada de dados. Ao invés disso, ele é tido como uma abstração desta, sendo pontuado fundamentalmente por ser um framework ORM. Na prática, sua utilização em dado projeto Delphi exige uma conexão efetiva com o banco de dados envolvido. Para isso, sua biblioteca de componentes contempla um elemento específico, denominado TEntityConnection, que assume então todo este direcionamento. Por definição, este elemento não é tido essencialmente como um componente de acesso a dados, sendo apenas uma interface para tal. Logo, toda sua mecânica gira em torno de sua estrutura interna, que faz uso de pacotes provenientes de outros elementos especializados para a realização das atividades diretas com a fonte de dados. Esses elementos referem-se aos próprios conjuntos de componentes de acesso a dados que fazem parte do ecossistema do Delphi, tais como os nativos (dbExpress, ADO, IBX e FireDAC) e de terceiros (AnyDAC, Zeos, UniDAC e etc.). Isso faz com que o EntityDAC obtenha a abstração necessária em termos de SGBDs suportados e tecnologias de acesso, estabelecendo-se como um framework consistente e flexível.

Internamente, a comunicação existente entre o TEntityConnection e componente de acesso a dados fica a cargo de um membro

intermediário, mais especificamente uma classe, tratada como provedor de dados (Data Provider). A fim de contribuir para o pleno entendimento deste cenário, toda a estrutura de acesso a dados do EntityDAC é então ilustrada na **Figura 6**.

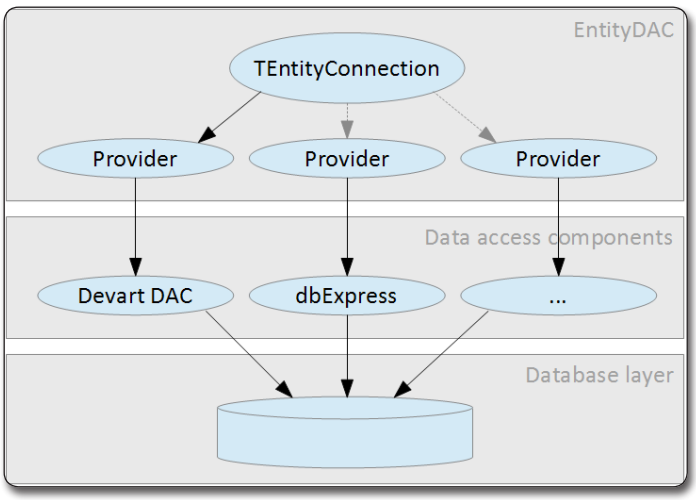

Figura 6. Estrutura de acesso a dados utilizada pelo EntityDAC – Fonte: site oficial do produto

Provedor de Dados (Data Provider)

Conforme mencionado, no contexto do EntityDAC, um Data Provider nada mais é do que uma classe que, visualmente, se transforma em um componente. Sendo assim, algumas opções de provider pré-definidas se mostram disponíveis na Tool Palette do IDE do Delphi, no grupo EntityDAC Providers, exclusivo da biblioteca do produto, conforme pode ser visto na **Figura 7**. Por se tratar de uma classe/componente, o uso de um provedor de dados numa aplicação EntityDAC requer seu registro, que ocorre tanto pela simples declaração de sua unit na seção Uses do código, ou pela adição de um componente no formulário em questão. Em termos funcionais, cada classe Data Provider implementa de forma concreta as operações de banco de dados relacionadas à determinada tecnologia de acesso a dados específica, e que são necessárias ao funcionamento do TEntityConnection.

Dialetos (SQL Dialects)

Um dos conjuntos de componentes de acesso a dados suportados pelo EntityDAC é o UniDAC (**BOX 1**), pertencente à mesma empresa fabricante do framework, a Devart. Como característica, esta biblioteca provê acesso a uma extensa gama de fontes de dados distintas, incluindo os principais SGBDs atuantes no cenário Delphi, tais como InterBase, Firebird, Oracle, MS SQL Server, MySQL, entre outros. De maneira semelhante, a ferramenta dispõe de suporte a bibliotecas mais específicas, direcionadas a um único tipo de banco de dados, como é o caso da opção nativa IBX (InterBase eXpress), voltada ao SGBD InterBase. Sendo assim, com o intuito de poder trabalhar da melhor forma com componentes de segmentos tão distintos, e atender às variações da linguagem SQL relacionadas a cada tipo de SGBD, o EntityDAC incorpora o conceito de SQL Dialects, que fundamentalmente define o dialeto exato a ser utilizado para a conexão com determinado banco de dados. No aspecto prático isso é refletido então numa propriedade denominada DialectName, presente no componente TEntityConnection. InterBase, Firebird, Oracle e SQL Server são alguns dos nomes de dialetos disponibilizados nativamente pelo componente, conforme ilustrado na **Figura 8**.

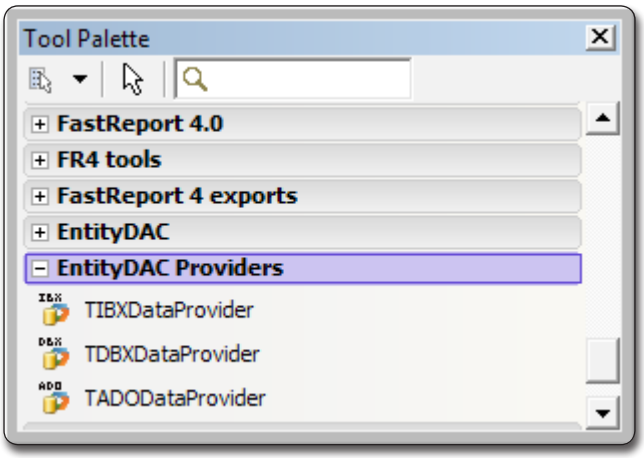

Figura 7. EntityDAC Providers

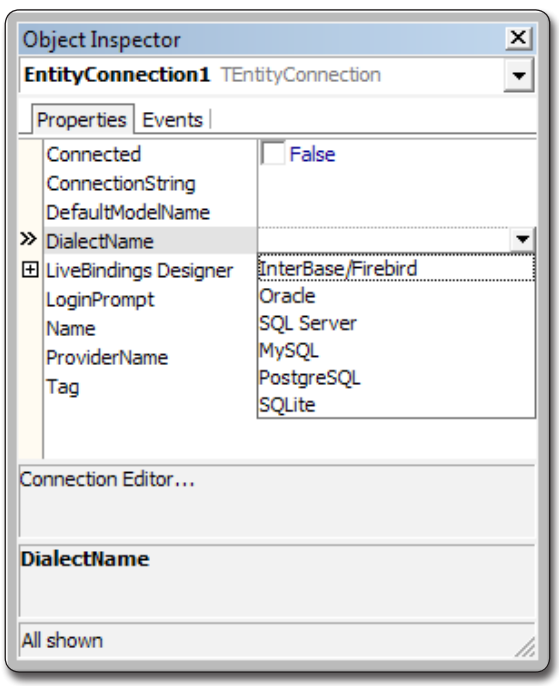

Figura 8. TEntityConnection – dialetos

BOX 1. UniDAC

Universal Data Access Components, ou simplesmente UniDAC, é uma biblioteca de componentes de acesso a dados provida pela empresa Devart, direcionada aos ambientes de desenvolvimento Delphi, Lazarus e C++ Builder. Como atrativo, possui capacidades suficientes para operar com diversas fontes de dados distintas, incluindo SGBDs como Firebird, InterBase, Oracle, Microsoft SQL Server, MySQL e DB2. Além disso, em termos funcionais, pode ser executado tanto em plataformas Desktop (Windows, Mac OS X, Linux e FreeBSD) quanto Mobile (iOS, Android).

String de conexão (Connection String)

Para cada conjunto de componentes de acesso a dados, a configuração de uma conexão a banco de dados envolve essencialmente a devida informação de parâmetros ao seu componente de conexão, tal como TSQLConnection para dbExpress e TIBDatabase para IBX. Obviamente, apesar de ambos estarem direcionados a um mesmo propósito, a configuração de cada um acaba por envolver características específicas. Assim, pelo fato do EntityDAC poder fazer uso de diferentes componentes de acesso a dados para lidar com a parte de conexão a banco, ele acaba por estabelecer um método unificado para este quesito, expresso por uma string de conexão (Connection String). Sendo assim, é nesta estrutura onde estarão indicados os parâmetros de conectividade que envolvem informações como o caminho do banco de dados, as credenciais de acesso (usuário e senha) e host condizente, independente da biblioteca de acesso a dados utilizada. Na prática, sua composição é determinada por uma cadeia de chave/valor separados por ponto e vírgula, conforme mostrado a seguir:

```
datasource=localhost;database=banco.FDB;userid=SYSDBA;password=masterkey
```

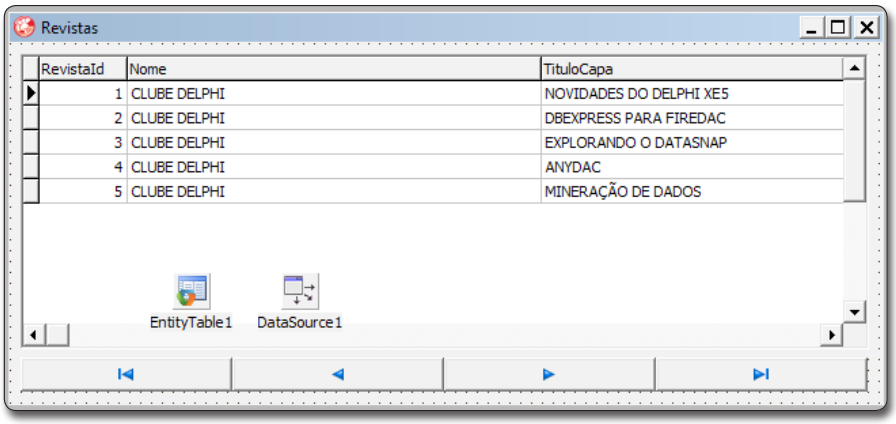

Figura 9. Formulário em modo Design – uso de TEntityTable, TDataSource e TDBGrid

Contexto (TEntityContext)

TEntityContext é o nome da classe presente no framework que provê as funções necessárias para o devido gerenciamento do ciclo de vida das instâncias de entidades (objetos) pertencentes ao cenário de uma aplicação EntityDAC. Dada sua importância e consequente necessidade de uso em aplicações deste tipo, sua disponibilidade ao desenvolvedor é refletida em um meio visual, através de um componente. Na prática, para o ideal provimento de suas funcionalidades, ele se liga a outros dois tipos de elementos pertencentes ao leque do produto, por meio de suas propriedades Connection e ModelName. Conforme a própria denominação que cada uma sugere, a primeira refere-se a um elemento de conexão TEntityConnection, enquanto que a segunda ao nome do modelo envolvido.

Recuperando dados da tabela (TEntityTable)

TEntityTable pode ser considerado o componente mais simples, dentre os disponibilizados na biblioteca do EntityDAC, para ser usado na recuperação de dados de uma tabela do banco de dados. Isto porque, assim como outros componentes Table do cenário tradicional, tal como o TSQLTable do dbExpress, seu uso não envolve a necessidade de escrita de instruções de consulta para obtenção dos dados. Ao invés disso, ele se liga diretamente a um determinado elemento da fonte de dados, fazendo todas as tratativas necessárias de forma automática e transparente ao desenvolvedor. No caso do TEntityTable, ele se liga a um único meta-type presente no Modelo por meio de sua propriedade TypeName, o que acaba por envolver a atribuição de um objeto do tipo TEntityContext em sua propriedade Context. De acordo com as intenções do exemplo, a configuração de um TEntityTable se daria basicamente da seguinte forma:

```
object EntityTable1: TEntityTable
  Context = DataModule1.EntityContext1
  TypeName = 'Revista'
  ...
end
```

Em complemento, os dados obtidos por este componente Table podem então ser direcionados a qualquer controle de visualização de dados, tal qual um TDBGrid, por meio de um TDataSource tradicional, conforme mostrado no código DFM a seguir em conjunto com a **Figura 9**:

```
object DataSource1: TDataSource
  DataSet = EntityTable1
  ...
end
```

Usando LINQ para consultar dados

Tomando partido de um cenário tradicional, a consulta a dados em um projeto de uma aplicação Delphi de banco de dados resume-se ao uso da SQL. Logo, numa exemplificação clara, para se retornar as informações contidas na tabela REVISTA do banco de exemplo, a seguinte instrução poderia ser utilizada:

```
select * from REVISTA
```

Indo além, esta mesma instrução poderia conter outros acréscimos relacionados, por exemplo, à especificação e ordenação de dados, que na linguagem SQL se traduzem no uso da cláusula WHERE e ORDER BY, respectivamente. Em vista disso, a seguir é mostrada a instrução a ser utilizada para a filtragem de revistas cujo número da edição (campo NUMERO_EDICAO) seja a partir de 150, a serem retornados em ordem crescente:

```
select * from REVISTA
where NUMERO_EDICAO >= 150
order by NUMERO_EDICAO
```

No desenvolvimento tradicional, esse tipo de instrução é costumeiramente direcionado a um componente do tipo Query, ficando alocado diretamente em uma de suas propriedades, no formato de texto simples. Tomando como exemplo o TSQLQuery da biblioteca do dbExpress, a propriedade em questão é denominada SQL e uma configuração padrão pode ser vista a seguir:

```
object SQLQuery1: TSQLQuery
...
  SQL.Strings = (
    'select * from REVISTA'
    'where NUMERO_EDICAO >= 150'
    'order by NUMERO_EDICAO')
...
end
```

Com o uso do EntityDAC, todo este contexto se altera, uma vez que os dados provindos do banco de dados não são interpretados como simples informações, ao invés disso, são transformados em instâncias de entidades de negócio (objetos). Consequentemente, o uso da SQL para a consulta a dados se torna naturalmente inapta para a conjuntura, sendo necessário outro tipo de abordagem. Para tal, conforme já mencionado anteriormente, o framework opera com uma linguagem própria, denominada LINQ, que acaba por estabelecer novas práticas e recursos de desenvolvimento. A primeira delas está no fato de que para usar o LINQ, o desenvolvedor irá lidar com código Delphi ao invés de simples strings. Por este fator, o uso do recurso de Code Completion provido pelo IDE torna-se habilitado, facilitando a escrita das instruções.

Com o intuito de se estabelecer a forma de uso do LINQ em aplicações EntityDAC, a seguir é expressa a dinâmica de formação de uma consulta similar a apresentada anteriormente, porém nas diretrizes da linguagem do framework.

From

Em uma instrução de consulta na linguagem SQL, a declaração FROM fundamentalmente especifica a fonte dos dados que dará origem ao resultado a ser retornado. Neste tipo de chamada, a posição do FROM se dá imediatamente após a definição de campos, tal como pode ser visto na exemplificação anterior. Tradicionalmente, as fontes de informação utilizadas são determinados tipos de elementos de um banco de dados, tais como Tabelas e Visões (Views), que podem ser pontuados como dois dos mais habituais. Seguindo por esta linha, o uso do FROM no LINQ também determina a especificação da fonte de dados da Query, contudo, o grande diferencial aqui é que ele caracteriza o ponto inicial de uma consulta, conforme se segue:

```
DataContext.From(Revista)
```

Neste breve demonstrativo, o código apresentado provê uma equivalência ao "from Revista" da SQL, do exemplo de estudo de caso atual. Além disso, neste novo cenário, o termo "Revista" não está relacionado diretamente à tabela do banco de dados, mas sim fazendo menção a uma coleção de entidades do tipo Revista. Adicionalmente, da mesma forma que na SQL, é possível ainda a partir do FROM, fazer a combinação de N fontes de dados, tal como a seguir:

```
DataContext.From(Revista).From(Autor)
```

Select

Na conjuntura da linguagem SQL, a declaração SELECT é utilizada para se obter um conjunto de registros (dados) oriundos de uma ou mais tabelas do banco de dados. Em termos conceituais, esse retorno acaba por caracterizar um resultado, que recupera zero ou N linhas de determinados elementos contidos na base. De forma semelhante, sob o aspecto do LINQ no EntityDAC, SELECT é o nome dado a uma expressão ou matriz de expressões, que irá especificar os campos de dados que irão compor o resultado de uma query. Contudo, o diferencial aqui fica por conta do tipo de retorno, que ao invés de dados simples, irá conter elementos formados, expressos sob a tutela de entidades. Pelo fato de envolver uma dinâmica totalmente distinta da tradicional utilização da SQL, a seguir é mostrada a formatação de uma instrução simples, que caracteriza um "SELECT" no contexto LINQ:

```
DataContext.From(Revista).Select(Revista['Nome'])
```

De acordo com o exemplo, o código apresentado possui a mesma funcionalidade da instrução "select NOME from REVISTA". Em complemento, a omissão do argumento relativo ao nome de determinado campo faz com que o resultado abranja todos os campos envolvidos, tal qual numa chamada a "select * from REVISTA", tal como utilizado no presente estudo de caso:

```
DataContext.From(Revista).Select
```

Where

Na SQL, o uso da cláusula WHERE determina a imposição de um critério de busca que irá especificar um determinado conjunto de informações de retorno que exclusivamente atendam aos requisitos restritivos declarados. Assim, em termos gerais, seu uso age como uma espécie de filtro de dados, sendo portanto naturalmente considerada uma cláusula opcional posicionada, tradicionalmente, após uma cláusula de origem de dados, tal qual o FROM. No LINQ, todo este cenário se repete, sendo a definição do WHERE baseada em uma expressão lógica que define uma determinada condição a qual os eventuais elementos resultantes da operação deverão atender. Por conseguinte, mediante a instrução SQL de exemplo apresentada anteriormente, sua representação no contexto LINQ se daria da seguinte forma:

```
DataContext.From(Revista)
    .Where(Revista['NumeroEdicao'] >= 150)
    .Select
```

Order By

Em termos práticos, na SQL, o uso da cláusula ORDER BY estabelece um critério de ordenação para as informações resultantes de uma consulta. Por este fato, assim com o WHERE, ela é tida como um membro opcional para a escrita de uma instrução. Quando usada, seu posicionamento padrão se dá após uma cláusula de fonte ou de filtro, tais como FROM e WHERE, respectivamente. No LINQ novamente todo este cenário se repete. Sem mais delongas, a seguir é então exposta uma amostragem de uso do ORDER BY no contexto da linguagem de consulta do EntityDAC, complementando a analogia feita com a SQL de exemplo abordada:

```
DataContext.From(Revista)
    .Where(Revista['NumeroEdicao'] >= 150)
    .OrderBy(Revista['NumeroEdicao'])
    .Select
```

Obviamente o contexto LINQ no EntityDAC é muito vasto, o que acabaria por exigir um artigo todo dedicado ao assunto. Apenas para citar, diversas outras cláusulas estão disponíveis em sua atmosfera, tais como Join (LeftJoin, RightJoin e FullJoin), GroupBy, ThenBy, Concat, Intersect, Skip, Take, Any, Count, Max, Min, entre outros, que o tornam robusto o suficiente para atender aos mais variados casos e cenários de aplicações Delphi que envolvam dados. Por fim, de forma conclusiva, é possível apontar o uso do LINQ como muito intuitivo, devido a sua proximidade com a SQL. Todavia, no cotidiano, o pleno entendimento sobre suas nuances de utilização requer certo tempo de prática diária. Prova disso é sua forma de escrita invertida, se comparada a da SQL, no que diz respeito aos membros que compõem a sintaxe da instrução.

De uma forma geral, o uso da Programação Orientada a Objetos no desenvolvimento Delphi é algo recorrente e até certo ponto natural, muito em razão pelo fato da própria linguagem seguir estritamente o referido paradigma. Assim, mesmo o desenvolvedor mais desatento, acaba por utilizar nuances da OO em suas aplicações, de forma congênita e transparente. Em complemento a este cenário, surgem as aplicações cuja fundação e consequente desenvolvimento se baseiam puramente nos aspectos da POO. Para estes casos, essencialmente para contextos envolvendo bancos de dados relacionais, é trazido à tona a necessidade pelo uso de "ferramentas" auxiliares, tal como o EntityDAC, para a parte de mapeamento objeto-relacional.

Autor

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ e pós-graduado em Engenharia de Componentes. Atua como Analista Programador Delphi há 9 anos. Atualmente escreve artigos sobre Delphi para revistas especializadas. Certified Delphi Developer. Certified Embarcadero Instructor. Site pessoal: www.fabriciokawata.com.br

Links:

RAD Studio XE5 - Página oficial do produto
http://www.embarcadero.com/products/rad-studio

RAD Studio XE5 - Download Trial
https://downloads.embarcadero.com/free/rad_studio

EntityDAC - Página oficial do produto
http://www.devart.com/entitydac

Firebird - Página oficial do projeto
http://www.firebirdsql.org

.NET Provider for Firebird
http://www.firebirdsql.org/en/net-provider

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

DAO Pattern: Persistência de Dados utilizando o padrão DAO

O Delphi é uma linguagem de programação, um IDE e um compilador. No que diz respeito a linguagem, ela implementa todos os requisitos de uma linguagem orientada a objetos, nos permite fazer abstração, encapsulamento, herança e polimorfismo.

A linguagem tem a característica de ser multiparadigma, ou seja, tem suporte a mais de um paradigma de programação (**BOX 1**), sendo possível o desenvolvimento estruturado e orientado a objetos. O que ocorre é a programação mista destes dois estilos de desenvolvimento.

Porém, o que mais vemos nos desenvolvedores Delphi é o desenvolvimento de aplicações de forma estruturada, onde são explorados apenas os recursos RAD que o IDE nos proporciona, e que são excelentes, sem dúvida, mas não são tudo no desenvolvimento de um sistema, devemos levar em consideração diversos outros aspectos não somente na agilidade de desenvolvimento.

Outro grande problema encontrado em sistemas escritos em Delphi é a mistura de lógica de negócios, lógica de acesso a dados e interface com usuário, onde estas geralmente estão totalmente acopladas, tornando extremamente difícil a manutenção dos sistemas e tornando o sistema vulnerável.

Fique por Dentro

Neste artigo aprenderemos como desenvolver um controle de acesso e cadastro de usuários orientado a objetos, separando o sistema em três camadas lógicas, a camada de lógica de negócios, a camada de acesso a dados e a camada de visualização. Veremos qual a maneira correta delas interagirem entre si no desenvolvimento de um sistema e também veremos várias boas práticas de orientação a objetos, tanto em nossas classes de negócio e acesso a dados, quanto nos formulários Delphi, onde muitas vezes deixamos isto de lado. Aprenderemos a utilizar exceções da maneira correta. Veremos como podemos criar formulários que obedecem aos principais princípios de orientação a objetos, sem deixarmos de aproveitarmos o RAD que o Delphi nos proporciona.

BOX 1. Paradigmas de Programação

Paradigmas de Programação representam um modelo ou estilo de programação suportado por linguagens que agrupam certas características comuns. Os mais conhecidos são a programação funcional (Lisp e Haskell), programação estruturada (Pascal e C), programação orientada a objetos (C# e Java). Outras ainda possuem suporte a múltiplos paradigmas (PHP, Delphi).

SQL Server Express

Para o desenvolvimento do exemplo contido neste artigo, faremos uso do banco de dados da Microsoft SQL Server Express, que é um sistema de gerenciamento de banco de dados desenvolvido pela Microsoft muito avançado, gratuito e confiável. O SQL Server é um dos bancos de dados mais utilizados no mundo atualmente.

Para quem está acostumado a utilizar o Firebird uma vantagem do SQL Server é a possibilidade de criar campos auto incremento de forma nativa no banco de dados, sem a necessidade de utilização de Generators, Triggers e Stored Procedures, tornado mais simples o gerenciamento de campos auto incremento.

O SQL Server possui versões gratuitas e pagas sendo que a versão Express (gratuita) do banco de dados da Microsoft atende a grande maioria das necessidades dos desenvolvedores, sendo a versão paga voltada a grandes empresas que necessitam de recursos avançados e grande desempenho.

FireDAC

A tecnologia de acesso a dados que será utilizada é o FireDAC, que é uma suíte de acesso a dados que foi adquirida pela Embarcadero junto a UniDAC, e nos proporciona um grande número de classes e componentes de alta performance para atender as necessidades dos desenvolvedores no que diz respeito a banco de dados.

O FireDAC possui diversas vantagens e recursos interessantes como Live Data, Array DML, API Unificada, Tratamento de Erros

Unificado, Arquitetura Simplificada, Restauração Automática de Conexão com a Base de Dados e Suporte a codificações ANSI e Unicode.

Novidades VCL

Na versão mais recente do Delphi, a versão XE6 trouxe melhorias em diversos pontos do IDE, incluindo a VCL. A VCL recebeu melhorias de performance e estabilidade.

Podemos, a partir desta versão, evoluir nossas aplicações Windows com novos componentes de sensor e termos interfaces com estilo do Windows 7/8 com novos estilos (incluindo menus e bordas) e componentes de barra de tarefa do Windows.

Com o novo recurso de Tethering (**BOX 2**) também podemos estender nossas aplicações através de aplicações Mobile, chamando ações de nossas aplicações VCL.

BOX 2. Tethering

Tethering é um novo recurso presente na versão Delphi XE6, onde é possível que aplicações de diferentes plataformas possam interagir diretamente, sem a necessidade de uma aplicação servidora intermediária, desde que estejam na mesma rede. Desta maneira podemos ter aplicações Desktop e Mobile interagindo e trocando dados.

Padrão DAO (Data Access Object)

O padrão de projeto DAO surgiu com a necessidade de separarmos a lógica de negócios da lógica de persistência de dados. Este padrão permite que possamos mudar a forma de persistência sem que isso influencie em nada na lógica de negócio, além de tornar nossas classes mais legíveis.

Classes DAO são responsáveis por trocar informações com o SGBD e fornecer operações CRUD e de pesquisas, elas devem ser capazes de buscar dados no banco e transformar esses em objetos ou lista de objetos, fazendo uso de listas genéricas (**BOX 3**), também deverão receber os objetos, converter em instruções SQL e mandar para o bando de dados. Toda interação com a base de dados se dará através destas classes, nunca das classes de negócio, muito menos de formulários.

Se aplicarmos este padrão corretamente ele vai abstrair completamente o modo de busca e gravação dos dados, tornando isso transparente para aplicação e facilitando muito na hora de fazermos manutenção na aplicação ou migrarmos de banco de dados.

Também conseguimos centralizar a troca de dados com o SGBD (Sistema Gerenciador de Banco de Dados), teremos um ponto único de acesso a dados, tendo assim nossa aplicação um ótimo design orientado a objeto.

BOX 3. Generics (TList x TObjectList)

A Classe TList possibilita guardarmos coleções de qualquer tipo de dados, tanto primitivos quanto objetos. Já a classe TObjectList, que é uma especialização de TList pode armazenar somente objetos. Ela tem a vantagem de no momento de liberarmos uma instância dela da memória, todos os objetos contidos nela também são liberados, diferentemente das instâncias de TList, onde precisamos liberar cada objeto individualmente da memória.

Mapeamento Objeto Relacional

Atualmente o tipo de banco de dados que predomina é o relacional. Porém o modelo relacional e o modelo orientado a objetos possuem algumas diferenças conceituais que precisam ser mapeadas.

Isso surgiu porque precisamos salvar o estado de um objeto (atributos, herança, polimorfismo) em tabelas do banco de dados. O mapeamento básico pode ser feito através de conversões manuais ou através de frameworks de persistência. Ambas as abordagens possuem vantagens e desvantagens, a conversão manual nos dá uma melhor performance enquanto que frameworks de persistência não precisamos nos preocupar com escrita de SQL.

Em Delphi temos alguns frameworks de mapeamento objeto relacional, sendo os principais o DORM (open source) e o Aurelius (comercial). Estes frameworks geralmente nos possibilitam a configuração de persistência através de arquivos XML ou o recurso de annotations ou custom atributes (**BOX 4**) do Delphi.

No mapeamento objeto relacional básico o que temos é o mapeamento de classes para uma tabela do banco de dados, temos também o mapeamento de objetos da classe para registros da tabela e as propriedades da classe para colunas da tabela.

BOX 4. Custom Attributes

Custom Attributes em Delphi é um recurso existente desde a versão 2009 da IDE e são semelhantes aos Annotations presentes no Java e C#, onde juntamente com a RTTI (Run Type Information) nos possibilita a criação de frameworks de persistência e de validação. Para criarmos um custom attribute devemos criar uma classe que herda de TCustomAttribute:

```
type
  Entity = class(TCustomAttribute)
end
```

Posteriormente podemos fazer uso do atributo nas demais classes, bastando declarar a unit correspondente na seção uses do arquivo .pas:

```
[Entity]
TPessoa = class(TObject)
end;
```

Criação da Classe Modelo

Quando vamos iniciar um projeto orientado a objetos, precisamos primeiramente pensar na modelagem de classes do nosso sistema, diferentemente de quando estamos trabalhando de forma estruturada, onde o primeiro passo é a criação da base de dados.

Nosso exemplo irá se basear na abstração de um registro de usuários, algo bastante comum no dia a dia dos desenvolvedores, porém os programadores Delphi na sua maioria não aplicam boas práticas de programação e orientação a objetos no gerenciamento de usuários de um sistema.

Nossa classe usuário terá um atributo Id que será nosso identificador da classe sequencial e o nome do usuário. Temos também o atributo Login que deverá ser único para cada usuário do sistema.

O próximo atributo é a senha que deverá ter o mínimo de cinco caracteres. Por último temos o atributo Ativo que indica se este usuário está com acesso liberado o restrito no sistema, conforme **Listagem 1**.

Esta classe nunca deve fazer qualquer conexão com o banco de dados, executar consultas, fazer atualizações, muito menos interagir com os formulários. Estas responsabilidades são das outras classes do nosso sistema.

As propriedades Id, Nome e Login foram criadas para ler e escrever diretamente em seus atributos, diferentemente da propriedade Senha que possui uma validação de entrada de dados, implementada no método SetSenha. Observe que declaramos este método como virtual, esta diretiva nos possibilita no caso de termos uma especialização desta classe, mudar esta regra de validação de senha, conforme **Listagem 2**.

Listagem 1. Interface da Classe TUsuario

```
01.  TUsuario = class(TObject)
02.  private
03.    FId: Integer;
04.    FNome: string;
05.    FLogin: string;
06.    FSenha: string;
07.  protected
08.    procedure SetSenha(const Value: string); virtual;
09.  public
10.    constructor Create; overload;
11.    constructor Create(AId: Integer; ANome, ALogin, ASenha: string); overload;
12.    property Id: Integer read FId write FId;
13.    property Nome: string read FNome write FNome;
14.    property Login: string read FLogin write FLogin;
15.    property Senha: string read FSenha write SetSenha;
16.  end;
```

Listagem 2. Implementação da Classe TUsuario

```
01.  constructor TUsuario.Create(AId: Integer; ANome, ALogin, ASenha: string);
02.  begin
03.    Id := AId;
04.    Nome := ANome;
05.    Login := ALogin;
06.    Senha := ASenha;
07.  end;
08.
09.  constructor TUsuario.Create;
10.  begin
11.  end;
12.
13.  procedure TUsuario.SetSenha(const Value: string);
14.  begin
15.    if Length(Value) < 5 then
16.      raise EArgumentException.Create('A senha deve ter no mínimo 5 caracteres')
17.    else FSenha := Value;
18.  end;
```

Outro detalhe importante é o uso de dois construtores, os construtores são extremamente úteis para inicializarmos atributos de nossas classes, porém também mantemos um construtor padrão, sem argumentos. Podemos ter diversos construtores, desde que estes tenham parâmetros diferentes e sejam seguidos pela diretiva Overload.

O construtor sem argumentos não faz nenhuma operação e o construtor que recebe os dados preenche os atributos de nossa classe.

Já no método SetSenha (linhas 13 a 18), implementamos uma regra de negócio, onde toda e qualquer instância da classe usuário só pode possuir senha com o número de dígitos maior do que cinco. Estas regras devem estar sempre centralizadas em nossas classes de negócio, pois quando houver necessidade de mudança, é somente nesse ponto que será necessário fazer a alteração, permanecendo o restante do sistema intacto.

Para lançarmos a exceção fizemos uso da classe EArgumentException que deve ser utilizada sempre que recebermos um argumento inválido, devemos procurar lançar exceções específicas ao invés de criarmos somente exceções do tipo Exception, pois facilita muito a manutenção do sistema, tornando o mesmo mais claro e de fácil entendimento.

Criando o Banco de Dados

Para o desenvolvimento do banco de dados foi utilizado SQL Server Express. Podemos verificar o script de criação da base de dados do exemplo na **Listagem 3**.

Listagem 3. Script criação do banco de dados

```
01.  CREATE database artigodao;
02.
03.  CREATE TABLE [dbo].[Usuarios](
04.     [Id] [int] IDENTITY(1,1) NOT NULL,
05.     [Nome] [varchar](50) NOT NULL,
06.     [Login] [varchar](25) NOT NULL UNIQUE,
07.     [Senha] [varchar](25) NULL,
08.   CONSTRAINT [PK_Usuarios] PRIMARY KEY CLUSTERED ([Id] ASC))
```

Nossa tabela terá um Id do tipo integer sendo campo identidade, que será incrementado automaticamente pelo banco. Também serão criados os campos alfanuméricos de Nome, Login e Senha.

A criação do banco de dados irá nos resultar em um arquivo chamado artigodao.mdf no diretório MSSQL\DATA de instalação do SQL Server. No download do artigo haverá o arquivo da base de dados utilizada no exemplo.

Conexão com o Banco de Dados

Iremos encapsular a conexão com o banco de dados numa classe que fará uso do padrão Factory para criação de conexões. Assim, toda vez que necessitarmos de uma conexão com a base de dados, seja para fazer consultas, quanto fazer atualizações, faremos a chamada do método que nos retornará a conexão pronta para o uso, conforme **Listagem 4**.

Para conectarmos ao SQL Server com o DBExpress, devemos adicionar a unit correspondente ao banco de dados na seção uses de nosso código, neste caso Data.DBXMSSQL, conforme linha 6.

Existem diversas maneiras de fazermos a conexão a banco de dados pelo FireDAC, uma delas é através de uma única string de conexão, que será usada neste exemplo.

Listagem 4. Fábrica de conexões com SQL Server via FireDAC

```
01.  unit uConnectionFactory;
02.
03.  interface
04.
05.  uses
06.   FireDAC.Comp.Client, FireDAC.Stan.Def, FireDAC.Phys.MSSQL,   FireDAC.
      Stan.Error,
07.   FireDAC.Stan.Async, FireDAC.VCLUI.Wait, FireDAC.DApt;
08.
09.  type
10.   TConnectionFactory = class
11.   private
12.    class var FDConnection: TFDConnection;
13.   public
14.    class function GetConnection: TFDConnection;
15.   end;
16.
17.  implementation
18.
19.  { TConnectionFactory }
20.
21.  class function TConnectionFactory.GetConnection: TFDConnection;
22.  const
23.   STR_CONEXAO = 'DriverID=MSSQL;Server=IJCOINF17\SQLEXPRESS;
24.  Database=artigodao;User_name=sa;Password=pkvc82';
25.  begin
26.   FDConnection := TFDConnection.Create(nil);
27.   FDConnection.ConnectionString := STR_CONEXAO;
28.   FDConnection.Connected := True;
29.   Result := FDConnection;
30.  end;
31.
32.  end.
```

Para podermos devolver uma conexão para nossas classes que farão uso, o primeiro passo é declararmos uma variável do tipo TFDConnection na seção private e criarmos um objeto em memória, conforme linhas 12 e 25.

Na linha 23 temos a definição de uma constante que terá a string de conexão, onde cada parâmetro é dividido por ponto e vírgula. Em primeiro lugar temos o driver que será utilizado, onde nesse caso é o MSSQL por estarmos fazendo acesso ao SQL Server, cada banco de dados possui um driver diferente. Em seguida temos o servidor do SQL Server que é representado pelo nome da máquina seguido pela instância do banco de dados (SQLEXPRESS). Na sequência temos o nome do banco de dados, usuário e senha.

Note que tanto a variável FDConnection quanto o método GetConnection são estáticos, ou seja, não são variáveis nem método de instâncias e sim de classe, ou seja, para acessar o método GetConnection não precisamos de uma instância de TConnectionFactory.

Classe UsuarioDAO

O primeiro passo ao codificarmos nossa classe DAO é a obtenção da conexão através do construtor e destrutor da classe, mantendo esta referência durante todo o ciclo de vida da classe UsuarioDAO através da variável privada Connection, conforme linha 5. O construtor e destrutor da classe podem ser vistos na **Listagem 5**.

A primeiro método que iremos implementar na classe DAO é o de login do usuário no sistema. Para isso uma interface irá passar um Login e Senha para verificar se possui ou não acesso, caso não possua uma exceção personalizada será gerada. Podemos ver a implementação na **Listagem 5**.

Listagem 5. Método EfetuarLogin e Exception

```
01.  EAcessoNegado = class(Exception);
02.
03.  TUsuarioDAO = class
04.  private
05.   Connection: TFDConnection;
06.  public
07.   constructor Create;
08.   destructor Destroy; override;
09.   procedure EfetuarLogin(const ALogin, ASenha: string);
10.   // CRUD
11.   procedure Insert(AUsuario: TUsuario);
12.   procedure Update(AUsuario: TUsuario);
13.   procedure Delete(AId: Integer);
14.   function FindAll: TList<TUsuario>;
15.  end;
16.
17.  constructor TUsuarioDAO.Create;
18.  begin
19.   Connection := TConnectionFactory.GetConnection;
20.  end;
21.
22.  destructor TUsuarioDAO.Destroy;
23.  begin
24.   Connection.Free;
25.   inherited;
26.  end;
27.
28.  procedure TUsuarioDAO.EfetuarLogin(const ALogin, ASenha: string);
29.  const
30.   SQL = 'select count(*) from USUARIOS where LOGIN = :p1
31.    and SENHA = :p2';
32.  begin
33.   if not (Connection.ExecSQLScalar(SQL, [ALogin, ASenha]) > 0) then
34.    raise EAcessoNegado.Create('Acesso ao sistema negado!');
35.  end;
```

Na linha 1 temos a declaração da exceção que vamos lançar caso o usuário seja inválido. O método EfetuaLogin recebe dois parâmetros, ALogin e ASenha que serão testados pela classe, veja que a declaração não devolve para o cliente um boolean indicando se foi ou não feito o login com sucesso. Na abordagem orientada a objetos, levantamos uma exceção caso ocorra um problema de login, evitando qualquer tipo de violação do sistema.

Nota

Por convenção, os nomes das classes começam com o prefixo T, com exceção das Exceções, que começam com o prefixo E.

Na implementação do método temos uma constante com o SQL da busca do usuário na base de dados (linha 30). Assim como no DBExpress, no FireDAC inserimos parâmetros nas querys através de ":" mais o nome deste parâmetro.

Na linha 32 temos a utilização de um interessante método existente no FireDAC que nos possibilita o retorno de um único valor, através do método ExecSQLScalar, que recebe um comando SQL seguido de um array de parâmetros (caso existam), retornando um variant, que pode ser qualquer tipo de dado. Testamos o retorno verificando se não for maior que zero geramos a exceção (linha 33).

Na **Listagem 6** temos métodos Insert, Update e Delete possuem cada um seu SQL de execução, conforme linhas 2, 13 e 23, variando o tipo de comando e os parâmetros. A nomeação dos parâmetros fica a critério do desenvolvedor, não tendo nenhuma regra específica.

Listagem 6. Métodos Insert, Update e Delete

```
01.  procedure TUsuarioDAO.Insert(AUsuario: TUsuario);
02.  const
03.   SQL = 'insert into USUARIOS values (:p1, :p2, :p3)';
04.  begin
05.   with AUsuario do
06.   begin
07.    Connection.ExecSQL(SQL, [Nome, Login, Senha]);
08.   end;
09.  end;
10.
11.  procedure TUsuarioDAO.Update(AUsuario: TUsuario);
12.  const
13.   SQL = 'update USUARIOS set NOME = :p1, LOGIN = :p2, SENHA = :p3';
14.  begin
15.   with AUsuario do
16.   begin
17.    Connection.ExecSQL(SQL, [Nome, Login, Senha]);
18.   end;
19.  end;
20.
21.  procedure TUsuarioDAO.Delete(AId: Integer);
22.  const
23.   SQL = 'delete from USUARIOS where ID = :p1';
24.  begin
25.   Connection.ExecSQL(SQL, [AId], [ftInteger]);
26.  end;
```

Veja que nos métodos de Insert e Update não recebemos Strings com os dados do usuário e sim uma instância da classe TUsuario, tornando o método muito mais claro e elegante. Já para o método Delete, basta recebermos o campo chave para excluirmos o registro.

Em métodos de atualização (insert, update e delete) utilizamos o comando ExecSQL do FireDAC para aplicarmos as mudanças.

Para finalizarmos a classe TUsuarioDAO, temos a implementação do método FindAll na **Listagem 7** que deve retornar todos os usuários cadastrados no sistema. Porém estes usuários irão retornar na forma de uma lista de usuários e não na forma de um DataSet como estamos acostumados na forma RAD de trabalhar com o Delphi. Este método é mais extenso e necessita de uma maior explicação para o seu completo entendimento.

O método FindAll irá nos retornar uma lista de usuários, através de generics. Diferentemente dos métodos de atualização, para os métodos de consulta necessitamos percorrer os dados através de um DataSet, afim de devolvermos a lista.

Nota

O Delphi, nas suas últimas versões, oferece um mecanismo de detecção de vazamento de memória e exibe uma mensagem de erro ao sairmos de uma aplicação quando existem objetos que não foram liberados na memória. Isso é muito importante, pois quando estamos trabalhando orientados a objetos temos a criação de muitas classes, e como o Delphi não possui um coletor de lixo, esta funcionalidade ajuda e muito os desenvolvedores. Para ativá-la basta declarar no código do projeto: ReportMemoryLeaksOnShutdown := True;

Listagem 7. Método FindAll da classe TUsuarioDAO

```
01.  function TUsuarioDAO.FindAll: TList<TUsuario>;
02.  var
03.   FDQuery: TFDQuery;
04.   Usuarios: TList<TUsuario>;
05.  begin
06.   FDQuery := TFDQuery.Create(nil);
07.   try
08.    with FDQuery do
09.    begin
10.     Connection := Self.Connection;
11.     Open('select * from USUARIOS');
12.     Usuarios := TObjectList<TUsuario>.Create();
13.     while not FDQuery.Eof do
14.     begin
15.      Usuarios.Add(TUsuario.Create(FieldByName('ID').AsInteger,
16.       FieldByName('NOME').AsString, FieldByName('LOGIN').AsString,
17.       FieldByName('SENHA').AsString));
18.      FDQuery.Next;
19.     end;
20.    end;
21.   finally
22.    FDQuery.Free;
23.   end;
24.   Result := Usuarios;
25.  end;
```

Para executarmos a Query declaramos uma variável FDQuery e uma Usuarios que é uma lista de usuários, conforme linhas 3 e 4. Após criarmos o objeto FDQuery, configuramos a propriedade Connection com a conexão já existente na classe (linha 10). Logo após, podemos chamamos o método Open, já passando o comando SQL para ele, sem a necessidade de atribuirmos na propriedade SQL, como ocorria anteriormente no SQLDataSet.

Após a execução do comando de seleção, criamos uma lista de usuários (linha 12) e fazemos um laço de repetição no DataSet gerado pela consulta.

A linha 15 é a mais interessante deste método, pois nela a cada registro da tabela usuários, adicionamos um novo objeto TUsuario na lista através do método Add. Observe não precisamos de uma variável auxiliar para os objetos do tipo TUsuario, passamos a referência automaticamente para a lista. Através do método construtor criado na classe TUsuario, preenchemos todos seus atributos automaticamente, passando o respectivo nome na tabela Usuários.

Por fim, liberamos o objeto FDQuery (linha 22), que foi contornado por try/finally de forma que independente do que ocorrer na execução da query, liberamos ela da memória. Por último é devolvida a lista de usuários para o cliente que chamou o método FindAll (linha 24).

Interface de controle de acesso

Para o formulário de controle de acesso, criamos um novo formulário e configuramos a propriedade Name para frmAcesso, Position para poScreenCenter e BorderStyle para bsDialog, de forma a impedir o redimensionamento da janela e bloquear botões de maximizar e minimizar, sendo possível apenas a saída ou a efetuação do login.

As entradas de login e senha serão dos componentes da classe TLabeledEdit que possui internamente um label, que pode ser acessado através da propriedade EditLabel, facilitando a vida do programador. No edit da senha, mudamos a propriedade PasswordChar para *, de modo que a senha fique oculta ao ser digitada. Por último temos dois botões Ok e Cancelar, tendo um formulário semelhante ao da **Figura 1**.

Neste exemplo foi utilizado o estilo "Cyan Dusk" para termos este modelo de interface. Para alterarmos a aparência da aplicação basta acessarmos o menu Project/Options/Application/Appearance.

Sempre que criamos um novo formulário no Delphi, ele automaticamente cria uma variável global abaixo do final da declaração da classe referenciando a mesma, conforme **Listagem 8**. O Delphi faz isso para que a variável global TApplication da unit Vcl.Forms possa ter acesso a todos estes formulários no código do projeto.

Tal forma de implementação fere os princípios de encapsulamento, pois a variável frmAcesso estará disponível para todas as unidades da aplicação utilizar. Desta forma, em nossa implementação iremos eliminar esta variável.

Listagem 8. Variável global

```
01 var
02   frmAcesso: TfrmAcesso;
```

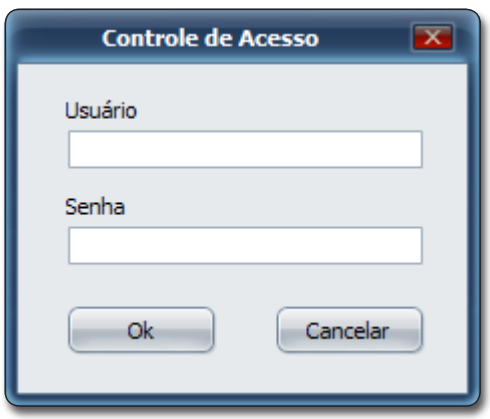

Figura 1. Formulário de Controle de Acesso

Nota

O Delphi nos permite a criação de variáveis globais porque é uma linguagem multiparadigma, ou seja, suporta tanto a programação estruturada quanto a programação orientada a objetos. Quando queremos trabalhar puramente orientado a objetos, variáveis globais não devem ser criadas.

Na seção public da classe declaramos uma propriedade somente leitura AcessoLiberado do tipo boolean, onde esta deverá ser consultada pelo formulário principal, afim de verificar se o usuário possui ou não acesso ao aplicativo, conforme **Listagem 9**. Esta propriedade é somente leitura, pois não faz sentido ela ter seu valor modificado externamente, somente este formulário deve ter a possibilidade de gravação nela, do contrário poderíamos acessá-la de outras unidades e violarmos o retorno, liberando acesso à aplicação, mesmo sem fazer a validação do usuário e senha. Observe que um único descuido pode quebrar completamente o encapsulamento e deixar a classe tolerante a falhas.

Na **Listagem 10** temos a implementação dos eventos onClick dos botões Ok e Cancelar. O Cancelar simplesmente fecha o formulário. Veja que não foi necessário atribuir False a nossa propriedade AcessoLiberado, isto porque atributos do tipo boolean são inicializados com False ao instanciarmos uma classe.

Listagem 9. Interface da classe TfrmAcesso

```
01.  TfrmAcesso = class(TForm)
02.    edtUsuario: TLabeledEdit;
03.    edtSenha: TLabeledEdit;
04.    btnOk: TButton;
05.    btnCancelar: TButton;
06.    procedure btnCancelarClick(Sender: TObject);
07.    procedure btnOkClick(Sender: TObject);
08.  private
09.    FAcessoLiberado: Boolean;
10.  public
11.    property AcessoLiberado: Boolean read FAcessoLiberado;
12.  end;
```

Listagem 10. Eventos OnClick dos botões Cancelar e Ok

```
01.  procedure TfrmAcesso.btnCancelarClick(Sender: TObject);
02.  begin
03.    Close;
04.  end;
05.
06.  procedure TfrmAcesso.btnOkClick(Sender: TObject);
07.  var
08.    UsuarioDAO: TUsuarioDAO;
09.  begin
10.    UsuarioDAO := TUsuarioDAO.Create;
11.    try
12.      try
13.        UsuarioDAO.EfetuarLogin(edtUsuario.Text, edtSenha.Text);
14.        FAcessoLiberado := True;
15.      except
16.        on E: EAcessoNegado do
17.          Application.MessageBox(PWideChar(E.Message), PWideChar(Self.Caption),
18.            MB_OK + MB_ICONERROR);
19.        on E: Exception do
20.          Application.MessageBox(PWideChar('Erro desconhecido' + #13 + E.Message),
21.            PWideChar(Self.Caption), MB_OK + MB_ICONERROR);
22.      end;
23.    finally
24.      UsuarioDAO.Free;
25.    end;
26.  end;
```

No evento de click do botão Ok temos a declaração da variável que irá manter uma referência a um objeto da classe TUsuarioDAO, em seguida temos a criação deste objeto em memória, conforme linhas X e Y. Após isso temos dois blocos try que nos auxiliarão a gerarmos uma exceção no caso de ocorrer um erro falha no acesso do usuário e um que servirá para liberarmos a instancia de TUsuarioDAO no final do método independente do que ocorrer durante a efetuação do login.

Veja que interessante as linhas X e Y, não foi necessário testar o retorno do método para sabermos se o usuário possui acesso ou não, esta é uma forma antiga e ultrapassada de fazermos validações. Imagine que temos várias formas de acesso a aplicação, como por exemplo, um acesso via uma aplicação web com IntraWeb ou via um Tablet com Android, se um dos desenvolvedores destes aplicativos clientes esquecer de testar este retorno irá liberar o acesso ao aplicativo, algo totalmente errado. Gerando uma exceção não temos mais este tipo de preocupação, pois a própria classe que faz o Login é a responsável por bloquear o acesso.

Se um erro acontecer durante a efetuação do login o compilador irá passar da linha 13 diretamente para o bloco except a partir da linha 15. Se a exceção for do tipo EAcessoNegado a mensagem padrão da exceção será exibida. Se ocorrer qualquer outro erro a mensagem irá inicial com Erro Desconhecido, seguido da mensagem de erro.

O tratamento do erro dentro do except se faz de forma sequencial, o compilador procura até encontrar o tratamento adequado para aquela exceção, se não se encaixa em nenhuma outra é gerada a classe base de todas as exceções do Delphi, a Exception.

Interface de cadastro de usuários

Da mesma forma que o controle de acesso, eliminamos a variável global criada automaticamente ao criarmos o novo formulário. Porém se formos tentar compilar neste momento a aplicação, teremos um erro, porque o objeto Application não consegue mais acessar a instância global de TfrmUsuario. Para resolvermos isso, basta declararmos a mesma no código do projeto através da opção Project/View Source, conforme linhas 15 e 16 da **Listagem 11**.

Nota

A propriedade MainFormOnTaskbar quando ativa mostra o nome da janela do formulário atual na barra de tarefas do Windows, quando inativa mostra o nome geral da aplicação, neste caso Artigo DAO.

Para exibirmos todos os usuários adicionamos um componente do tipo TStringGrid. Alinhamos alBottom, marcamos a opção RowSelect da propriedade Options para que seja selecionada sempre toda a linha, também deixamos em zero a propriedade FixedCols para que não tenhamos uma coluna fixa e mudamos para dois a propriedade ColCount pois exibiremos apenas duas colunas no StringGrid.

Para trabalharmos com os campos da classe adicionamos três componentes do tipo TLabeledEdit e configuramos a opção LabelPosition para lpLeft, para termos o rótulo no lado esquerdo do Edit. Usamos também um componente TStaticText para o título do formulário e três botões para as ações Cadastrar, Atualizar e Limpar campos.

Listagem 11. Código do Projeto

```
01.  program Exemplo;
02.
03.  uses
04.   Vcl.Forms,
05.   ufrUsuario in 'ufrUsuario.pas' {frmUsuario},
06.   uConnectionFactory in 'uConnectionFactory.pas',
07.   uUsuario in 'uUsuario.pas',
08.   uUsuarioDAO in 'uUsuarioDAO.pas',
09.   ufrAcesso in 'ufrAcesso.pas' {frmAcesso},
10.   Vcl.Themes,
11.   Vcl.Styles;
12.
13.  {$R *.res}
14.
15.  var
16.   frmUsuario: TfrmUsuario;
17.
18.  begin
19.   ReportMemoryLeaksOnShutdown := True;
20.   Application.Initialize;
21.   Application.MainFormOnTaskbar := False;
22.   TStyleManager.TrySetStyle('Cyan Dusk');
23.   Application.Title := 'Artigo DAO';
24.   Application.CreateForm(TfrmUsuario, frmUsuario);
25.   Application.Run;
26.  end.
```

Além destes, adicionamos um componente PopupMenu, onde adicionamos um item chamado Excluir, e vinculamos este componente na propriedade PopupMenu do String Grid.

Não foi a toa que criamos o método PopulaControles na seção Published. Fizemos isso para que conseguíssemos vincular este método diretamente no evento OnClick do StringGrid, através do Object Inspector, sem a necessidade de fazermos este vínculo programaticamente.

Conforme a **Listagem 12**, declaramos três objetos de nossa infraestrutura para trabalharmos no formulário (linhas 20, 21 e 22). O primeiro deles é o que representa um único Usuário, onde exibiremos seus dados nos Edits. Já segundo servirá para carregarmos todos os usuários do sistema e exibirmos na grade de dados, por último temos o UsuarioDAO que fornecerá as operações relativas à busca e atualizações no banco de dados.

Em nenhum momento teremos qualquer tipo de comando SQL no formulário, eles sempre estarão presentes única e exclusivamente na camada de acesso a dados (TUsuarioDAO).

No evento de abertura do formulário de usuários (OnShow) da **Listagem 13**, a primeira ação é verificarmos se o usuário possui acesso ao sistema, para isso chamamos o método VerificaAcesso (linha 3). Na sequência é chamado um método que configura os nomes das colunas do grid (linha 4), também criamos um objeto da classe TUsuarioDAO que permanecerá até fecharmos este formulário e por último carregamos os usuários já existentes na base de dados através do método CarregaGrid.

Listagem 12. Interface da classe TfrmUsuario

```
01.  TfrmUsuario = class(TForm)
02.    grdUsuarios: TStringGrid;
03.    StaticText1: TStaticText;
04.    edtNome: TLabeledEdit;
05.    edtLogin: TLabeledEdit;
06.    edtSenha: TLabeledEdit;
07.    btnCadastrar: TBitBtn;
08.    btnAtualizar: TBitBtn;
09.    btnLimpar: TBitBtn;
10.    PopupMenu1: TPopupMenu;
11.    Excluir1: TMenuItem;
12.    procedure FormShow(Sender: TObject);
13.    procedure btnLimparClick(Sender: TObject);
14.    procedure FormClose(Sender: TObject; var Action: TCloseAction);
15.    procedure btnCadastrarClick(Sender: TObject);
16.    procedure btnAtualizarClick(Sender: TObject);
17.    procedure Excluir1Click(Sender: TObject);
18.  private
19.    { Private declarations }
20.    Usuario: TUsuario;
21.    Usuarios: TList<TUsuario>;
22.    UsuarioDAO: TUsuarioDAO;
23.    procedure VerificaAcesso;
24.    procedure ConfigCaptionGrid;
25.    procedure CarregaGrid;
26.    procedure PreparaObjeto;
27.  public
28.    { Public declarations }
29.  published
30.    procedure PopulaControles(Sender: TObject);
31.  end;
```

O método VerificaAcesso irá criar uma instância do formulário de controle de acesso e abri-lo (linhas 13 e 15). Na volta do controle de acesso, testaremos a propriedade AcessoLiberado do frmAcesso, verificando se o usuário passou usuário e senha corretos, se forem incorretos encerramos o aplicativo (linha 17). Por último liberamos o formulário de acesso através do método Release.

Veja que não há nenhuma necessidade de mantermos aquela variável global criada pelo Delphi para o formulário de controle de acesso. O único ponto que utilizaremos este formulário é neste método, portanto declaramos a variável somente neste escopo (linha 11). Em nenhum outro ponto do sistema precisaremos acessar este formulário.

Nas linhas 23 a 27 temos a implementação do método ConfigCaptionGrid, que simplesmente atribui uma descrição para o título das colunas do StringGrid.

O método CarregaGrid serve para listarmos todos os usuários na grade de dados. Para isso chamamos o método FindAll (linha 35) da classe TUsuarioDAO e percorremos a lista de usuários retornada através do recurso de For/In, onde cada elemento da lista é atribuído a variável U, podendo nós preenchermos os dados do grid a cada iteração (linhas 36 a 42). Por último informamos ao StringGrid o total de linhas que desejamos que sejam exibidas. Como não estamos trabalhando com DataSets, o preenchimento da grade de dados deve ser de forma manual, como apresentado.

Observe a linha 38, nela estamos passando um objeto para cada linha do StringGrid, isto será muito útil quando formos recuperarmos os dados de um único objeto ao trocar de linha no StringGrid.

Listagem 13. Evento OnShow e métodos auxiliares do formulário

```
01.  procedure TfrmUsuario.FormShow(Sender: TObject);
02.  begin
03.    VerificaAcesso;
04.    ConfigCaptionGrid;
05.    UsuarioDAO := TUsuarioDAO.Create;
06.    CarregaGrid;
07.  end;
08.
09.  procedure TfrmUsuario.VerificaAcesso;
10.  var
11.    frmAcesso: TfrmAcesso;
12.  begin
13.    Application.CreateForm(TfrmAcesso, frmAcesso);
14.    try
15.      frmAcesso.ShowModal;
16.      if not (frmAcesso.AcessoLiberado) then
17.        Application.Terminate;
18.    finally
19.      frmAcesso.Release;
20.    end;
21.  end;
22.
23.  procedure TfrmUsuario.ConfigCaptionGrid;
24.  begin
25.    grdUsuarios.Cells[0, 0] := 'Nome';
26.    grdUsuarios.Cells[1, 0] := 'Login';
27.  end;
28.
29.  procedure TfrmUsuario.CarregaGrid;
30.  var
31.    U: TUsuario;
32.    i: Integer;
33.  begin
34.    i := 1;
35.    Usuarios := UsuarioDAO.FindAll;
36.    for U in Usuarios do
37.    begin
38.      grdUsuarios.Objects[0, i] := U;
39.      grdUsuarios.Cells[0, i] := U.Nome;
40.      grdUsuarios.Cells[1, i] := U.Login;
41.      Inc(i);
42.    end;
43.    grdUsuarios.RowCount := i;
44.  end;
```

Criamos alguns métodos auxiliares que ajudarão na tarefa de trabalharmos com os objetos. O primeiro é o PreparaObjeto (**Listagem 14**), nele atribuímos os dados digitados informados na interface nas propriedades do nosso objeto Usuário. O segundo método auxiliar PopulaControles faz o contrário, ele pega o Usuário atual selecionado no StringGrid e preenche os controles de tela, conforme **Figura 2**.

Na linha 13 da **Listagem 14**, o que fizemos recuperar o objeto que foi vinculado ao StringGrid e atribuirmos a nossa variável Usuário, para em seguida preenchermos os Edits.

O evento LimparClick simplesmente apaga os dados dos controles de tela através do método Clear para que o formulário esteja apto a cadastrar um novo usuário.

O evento Cadastrar primeiramente instancia nossa classe TUsuario (linha 28), em seguida chama o método PreparaObjeto para que suas propriedades sejam preenchidas com os dados informados na tela, para posteriormente chamar o método In-

sert da classe TUsuarioDAO, passando o objeto Usuário criado. A partir daí a classe TUsuarioDAO é quem vai prosseguir com a criação do comando SQL, obtenção da conexão e execução do comando. No final o método CarregaGrid é chamado para retornar o novo usuário cadastrado junto aos já existentes anteriormente na grade de dados. Veja que o método ficou simples e objetivo.

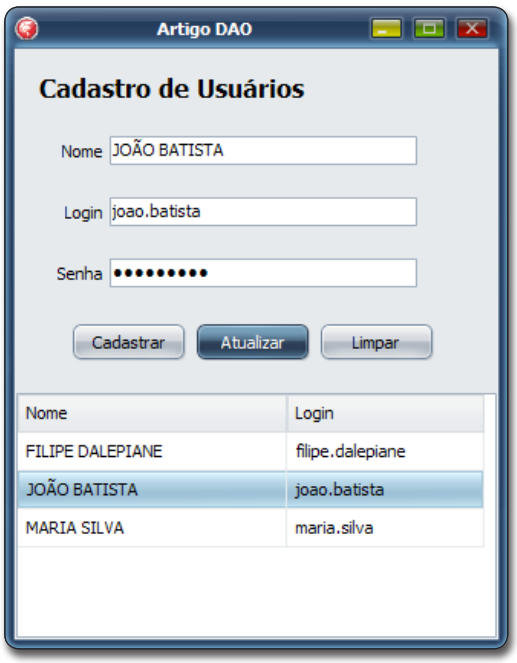

Figura 2. Interface do Cadastro de Usuários

O evento de AtualizarClick é ainda mais simples, ele pega o objeto que já foi escolhido na base de dados e simplesmente atualiza os dados das propriedades através do método PreparaObjeto, para no fim chamar o método Update da classe TUsuarioDAO. Depois da atualização novamente a grade de dados é atualizada para refletir os dados atuais que constam no banco de dados.

Para fechar temos o evento de exclusão, que foi implementado no PopupMenu, ao acionarmos o botão direito do mouse no StringGrid. Primeiro é solicitado a confirmação de exclusão do registro, para depois passarmos o identificador do objeto que está selecionado atualmente no grid. Como o StringGrid retorna tipos Object, foi necessário efetuar um typecast para TUsuario, para conseguirmos acessar a propriedade Id e passar para o método Delete da classe TUsuarioDAO (linha 45). Depois de efetuada a exclusão, carregamos novamente o grid para atualizar os dados (linha 46) e chamamos o método PopulaControles (linha 47) para carregar os dados do registro atual do grid.

Note que utilizamos a função Format para passarmos a mensagem de exclusão, que foi criada na forma de ResourceString (**BOX 5**) e passamos o argumento Nome da classe Usuário para completar a mensagem de solicitação de exclusão, conforme **Listagem 15** e **Figura 3**.

Listagem 14. Implementação dos eventos OnClick

```
01.  procedure TfrmUsuario.PreparaObjeto;
02.  begin
03.   with Usuario do
04.   begin
05.    Nome := edtNome.Text;
06.    Login := edtLogin.Text;
07.    Senha := edtSenha.Text;
08.   end;
09.  end;
10.
11.  procedure TfrmUsuario.PopulaControles;
12.  begin
13.   Usuario := TUsuario(grdUsuarios.Objects[0, grdUsuarios.Row]);
14.   edtNome.Text := Usuario.Nome;
15.   edtLogin.Text := Usuario.Login;
16.   edtSenha.Text := Usuario.Senha;
17.  end;
18.
19.  procedure TfrmUsuario.btnLimparClick(Sender: TObject);
20.  begin
21.   edtNome.Clear;
22.   edtLogin.Clear;
23.   edtSenha.Clear;
24.  end;
25.
26.  procedure TfrmUsuario.btnCadastrarClick(Sender: TObject);
27.  begin
28.   Usuario := TUsuario.Create;
29.   PreparaObjeto;
30.   UsuarioDAO.Insert(Usuario);
31.   CarregaGrid;
32.  end;
33.
34.  procedure TfrmUsuario.btnAtualizarClick(Sender: TObject);
35.  begin
36.   PreparaObjeto;
37.   UsuarioDAO.Update(Usuario);
38.   CarregaGrid;
39.  end;
40.
41.  procedure TfrmUsuario.Excluir1Click(Sender: TObject);
42.  begin
43.   if MessageDlg(Format(MsgExcluir, [Usuario.Nome]), mtConfirmation,
     [mbYes, mbNo], 0) = mrYes then
44.   begin
45.    UsuarioDAO.Delete(TUsuario(grdUsuarios.Objects[0, grdUsuarios.Row]).Id);
46.    CarregaGrid;
47.    PopulaControles(Self);
48.   end;
49.  end;
```

Listagem 15. ResourseString para mensagem de exclusão

```
1. resourcestring
2.  MsgExcluir = 'Deseja realmente excluir %s ?';
```

Figura 3. Confirmação de Exclusão

BOX 5. ResourceString

ResourceString são cadeias de caracteres que são vinculadas aos executáveis na forma de recursos, de forma que seus dados podem ser modificados sem a necessidade de recompilar o programa. Elas são idênticas as constantes, exceto por possuir esta característica citada. ResourceString podem ser declaradas em qualquer parte de uma unit do Delphi, desde que seja declarada antes de ser utilizada em um método ou função, isto porque o compilador (DCC32.EXE) é top down. Sua utilização é muito útil no desenvolvimento de sistemas multi-idiomas, onde podemos substituir os dados de todas as ResourceString para refletir o idioma selecionado em alguma configuração do sistema.

Como vimos, pode-se utilizar todo o poder do RAD presente no Delphi sem que violemos as boas práticas de programação e orientação a objetos.

Ao escrevermos sistemas bem modularizados seguindo os bons princípios da orientação a objetos e padrões de projeto, temos a garantia de um software mais seguro, mais claro e pronto para extensões que não irão afetar em toda a estrutura.

A primeira vista um sistema escrito seguindo todos os princípios da orientação a objetos parece mais difícil e complexo, mas as principais vantagens desta abordagem são percebidas a médio e longo prazo, onde surgem as necessidades de alterações e melhorias nos sistemas.

Autor

Filipe Dalepiane
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/

Analista Desenvolvedor de Sistemas numa Universidade.Colunista das revistas Clube Delphi e .NET Magazine.Desenvolvedor Delphi e C#, utilizando bancos de dados SQLServer e Firebird. Bacharel em Ciência da Computação.

Links:

Novidades do Delphi XE 6
https://www.embarcadero.com/br/products/delphi

Core J2EE Patterns - Data Access Object
http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

CURSOS ONLINE

A Revista SQL Magazine oferece aos seus assinantes uma série de Cursos Online de alto padrão de qualidade.

Feita para Desenvolvedores de Software e DBAs
SQL magazine

CONHEÇA OS CURSOS MAIS RECENTES:

- **Cursos: Curso de noSQL (Redis) com Java**
- **Desenvolvimento para SQL Server com .NET**
- **Curso PostgreSQL - Treinamento de banco de dados (Curso Básico)**

Para mais informações :
http://www.devmedia.com.br/cursos/banco-de-dados
(21) 3382-5038

DEVMEDIA

FireDAC CRUD: Cadastro completo utilizando VCL e FireMonkey

O Delphi desde o seu surgimento em 1994 tem se caracterizado por ser a ferramenta mais produtiva para o desenvolvimento de aplicações com banco de dados. Devido ao seu estilo RAD e algumas classes muito bem projetadas como a TDataSet e o TField, fez com que se tornasse uma ferramenta praticamente imbatível nessa área.

Quando surgiu o Delphi, o principal conjunto de componentes de acesso a dados era o BDE (Borland Database Engine). Na versão do Delphi 6 surgiu a suíte de componente chamada de dbExpress, que até a versão XE2, era a principal engine de banco de dados no Delphi. Porém, a partir da versão XE3 surgiu o FireDAC, que trouxe uma grande evolução neste quesito.

Recomenda-se fortemente que novos projetos iniciados com o Delphi façam uso do FireDAC para o acesso a dados. O BDE e o dbExpress ainda estão presentes no conjunto de componentes do Delphi, porém, não serão mais evoluídos com a ferramenta.

Fique por Dentro

Neste artigo teremos uma visão da evolução de acesso a dados no Delphi. Faremos o desenvolvimento de um CRUD básico com FireDAC através de um cadastro que pode servir de modelo para a criação de todo um sistema. Também teremos uma visão das características do FireDAC, seus principais componentes, o porquê que ele deve ser adotado para novos projetos com banco de dados e a razão de considerarmos a migração de sistemas existentes com outros mecanismos para o FireDAC. Aprenderemos algumas boas práticas para a confecção de interfaces e também dicas para o desenvolvimento de aplicações Client-Server. Também veremos como utilizar um mesmo DataModule para aplicações VCL e FireMonkey. No fim, converteremos nosso formulário FireMonkey para o formato Metropolis UI, a plataforma gráfica do Windows 8.

CRUD

CRUD é o acrônimo para Create, Read, Update, Delete. Ou seja, são as quatro principais operações utilizadas nos bancos de dados relacionais. Representam respectivamente as instruções de Insert, Select, Update e Delete.

FireDAC

FireDAC é o nome dado a versão da Embarcadero após a aquisição da suíte AnyDAC, que foi integrada tanto ao Delphi quanto ao C++ Builder. No Delphi, o FireDAC está disponível a partir da versão XE3 através de um instalador a parte, já na versão XE5, a suíte encontra-se presente na instalação padrão.

Trata-se de um conjunto de componentes de alta performance e fácil utilização que provêm conexão com diferentes bancos de dados, tanto locais quanto corporativos. Cada banco de dados possui um driver e o FireDAC trabalha com cada um deles de forma diferente. Os drivers do FireDAC são nativos para cada banco em particular e ainda possuem integração aos tipos ODBC e dbExpress.

Para quem trabalha com a arquitetura multicamadas com Delphi e DataSnap (**BOX 1**), ou servidores REST (**BOX 2**), basta migrar a camada relativa ao Servidor de aplicações de dbExpress para FireDAC, pois, o componente TFDQuery é um TDataSet, portanto, é compatível com o TDataSetProvider. Na camada Cliente, o TClientDataSet permanece sem necessidade de qualquer alteração.

BOX 1. DataSnap

O DataSnap é o principal Framework para o desenvolvimento multicamadas no Delphi. Servidores de aplicação DataSnap podem ser desenvolvidos tanto em Delphi quanto em C++ Builder, porém, a grande vantagem está na aplicação Cliente, que pode ser desenvolvida em qualquer linguagem de programação que possua suporte à JSON. O Delphi 2009 trouxe uma grande mudança na arquitetura do DataSnap. A partir desta versão a tecnologia COM foi abolida, mas algo tinha que substituí-la. Neste caso, como o Delphi já possuía uma tecnologia de acesso a dados muito eficiente, totalmente orientada a objetos e 100% escrita na própria linguagem, o dbExpress, a solução adotada foi criar um novo driver voltado ao DBX, no entanto, sua responsabilidade não seria a de se conectar a um banco de dados, mas sim, a um servidor de aplicação. Assim, a partir do Delphi 2009, a conexão do cliente com o servidor de aplicação passou a ser realizada utilizando o dbExpress trafegando os dados utilizando protocolo TCP/IP.

BOX 2. REST

REST é a abreviação de Representational State Transfer ou transferência de estado representacional. Teve seu surgimento a partir da tese de doutorado sobre a Web escrita por Roy Fielding, que foi um dos idealizadores do protocolo HTTP. Alguns entendem que o REST é um novo protocolo, outros o chamam de arquitetura e alguns defendem que ele é apenas uma filosofia, uma forma de pensar, um estilo de projetar uma aplicação. O REST é uma boa prática, onde o principal objetivo é construir aplicações fracamente acopladas, voltadas principalmente para o ambiente Web.

O FireDAC é um framework que possui um conjunto de classes e componentes TDataSet, semelhante aos engines tradicionais do Delphi. Além disso, este possibilita o desenvolvimento de aplicações para múltiplos banco de dados de forma RAD, algo que o dbExpress tentou fazer, porém era complicado, afinal tínhamos de que fazer várias configurações como, por exemplo, o mapeamento de campos TFields de acordo com cada banco utilizado. Uma das principais vantagens do FireDAC é a sua alta performance de acesso e a enorme quantidade de recursos e configurações disponíveis.

Os bancos de dados na qual conectam-se de forma nativa são os seguintes:

- MySQL
- Microsoft SQL Server
- Oracle Database
- InterBase
- PostgreSQL
- DataSnap
- SQLite
- Sybase SQL Anywhere
- Microsoft Access
- IBM DB2 Server
- Firebird
- Advantage Database

Características

É possível elencar alguns itens que ganham destaque no FireDAC:

- Live Data - Tabelas que podem ser editadas em tempo real, útil para tabelas menores ou bancos de dados locais. Permite navegar em DataSets de modo bidirecional;
- Array DML - São Arrays de scripts. Podem ser feitos vários Inserts, Updates ou Deletes em uma única requisição, tornando a aplicação mais performática e reduzindo o tráfego na rede. Ótimo para ambientes com rede lenta;
- API unificada - Padrão nos comandos SQL, controle de transações e erros unificados. Capacidade de busca de metadados melhoradas.
- Tratamento de erros unificado - O FireDAC interpreta os erros gerados por cada banco de dados e os transforma em mensagens padrão para retorno.
- Arquitetura simplificada - Pode-se ter um cursor bidirecional apenas com o componente TFDQuery, sem a necessidade de usar (TSQLQuery, TDataSetProvider, TClientDataSet) como no dbExpress;
- Recuperação automática de conexão ao banco de dados ao perder a conexão em caso de instabilidade do sistema;
- Suporte a codificações ANSI e Unicode.

Além dos itens destacados, também podemos citar o FireDAC Connection Editor do FireDAC, que é o editor de conexão disponível. Este é bem avançado em relação aos outros que os programadores Delphi estavam acostumados e provê uma grande quantidade de configurações e parametrizações, desde básicas até avançadas. Estas configurações podem ser salvas em um arquivo .INI que posteriormente pode ser distribuído e/ou alterado. Ainda sobre editores, o editor de Querys do FireDAC é muito superior ao que existia no SQLQuery do dbExpress. Pode-se com ele testar as Querys desenvolvidas, configurar os parâmetros, tipo de retorno e muitas outras opções.

Temos ainda o mecanismo Tracing para envio de log de dados para um arquivo de texto ou para um servidor remoto. Se a aplicação possui erros, pode-se analisar os logs gerados pelo FireDAC para facilitar a identificação do problema.

Existe ainda uma grande quantidade de opções de configuração de ambiente:

- FetchOptions - Controle de requisições, forma como serão trazidos os dados e quantidade de vezes que o FireDAC acessará o banco para buscar estas informações;
- FormatOptions - Formatação dos dados, podendo padronizar o tratamento e torna-los multi banco;
- ResouceOptions - monitoramento dos recursos utilizados;
- UpdateOptions - configuração de como os Updates serão gerados. O FireDAC gera estas instruções automaticamente;
- Rowset Fetching: há a possibilidade de limitar o número de registros que serão trafegados na rede. Este recurso é muito importante em redes lentas;
- Conexão Offline: permite que a aplicação continue a funcionar mesmo sem uma conexão física com o banco de dados.

Processamentos demorados podem ser feitos de modo assíncrono, podendo o usuário realizar outras tarefas enquanto este é efetuado. Existe um evento que será chamado no final desta tarefa e que poderá alertar o término para a exibição de alguma mensagem para o usuário.

É possível também definir um timeout para as consultas, ou seja, se a execução de uma Query atingir um limite de tempo determinado pelo desenvolvedor, ela será abortada automaticamente. Isso evita de haver travamentos de banco de dados, quando o usuário não determina os parâmetros para alguma consulta, por exemplo.

Existem funções avançadas no FireDAC que podem ser utilizadas quando desejamos personalizar um SQL para determinado banco. Este recurso é extremamente útil quando temos uma função nativa de um determinado banco de dados e não existente para outro. Exemplo:

```
Select * from CLIENTES
{ IF DB2 } fetch first 100 rows only {fi}
```

No exemplo estamos informando ao Framework que caso o banco seja o DB2, deve-se adicionar o comando "fetch first 100 rows only" ao final da consulta.

A geração de comandos SQL é feita de forma automática pelo FireDAC, considerando o banco de dados em uso na aplicação. No entanto se o desenvolvedor deseja fazer os comandos manualmente, o FireDAC disponibiliza o componente TFDUpdateSQL.

Além disso, há o recurso de detecção de campos auto incremento, onde são identificados recursos nativos como o Generator, usando-os para geração dos comandos SQL. Há um recurso que permite também a execução de SQL local, diretamente em DataSets através do componente TFDLocalSQL.

O FireDAC disponibiliza uma gama muito grande de recursos extras, como por exemplo: suporte a múltiplas transações, relatórios e logs de erros, suporte a eventos (se o banco suportar), backup, restore, validações de banco de dados e pooling de conexão, permitindo obter maior performance e compartilhamento de conexões com os usuários.

A **Figura 1** ilustra toda a arquitetura do FireDAC.

Principais Componentes do FireDAC

A seguir uma breve descrição dos principais componentes.

• FDConnection - provê a conexão com o banco de dados através de um editor de conexões;

• TFDTransaction - componente para trabalhar com transações;

• TFDMemTable - Dataset de memória, semelhante ao TClientDataSet, porém, com uma melhor performance. Utilizado para armazenamento de dados em cache. Possui recursos de ordenação, filtragem, agregação, índices filtrados e de expressão, persistência e etc.;

• TFDQuery - componente para execução de Querys. Também possui internamente uma cache de dados que pode ser ativada se desejado. Semelhante ao SQLDataSet do dbExpress, com a vantagem de possuir integrada a cache de dados, dispensando o uso de outro componente para esta tarefa;

• TFDUpdateSQL - componente que permite a criação manual de Querys de Insert, Update, Delete e etc., substituindo os comandos automáticos do TFDQuery;

• TFDStoredProc - componente para execução de Stored Procedures;

• TFDTable - componente para navegação e edição diretamente na tabela;

• TFDLocalSQL - componente para execução de consultas em DataSets locais;

• TFDScript - componente para execução de Scripts em lote;

• TFDSchemaAdapter - componente que server para concentrar o cache de DataSets.

Criação do Banco de Dados

O exemplo apresentado no artigo fará uso do banco de dados Firebird, muito conhecido e utilizado pela comunidade Delphi. Criaremos além do banco, uma tabela, uma trigger e uma Sequence. Para isso, faremos uso do utilitário ISQL que vem instalado junto do Firebird. O script completo pode ser visualizado na **Listagem 1**.

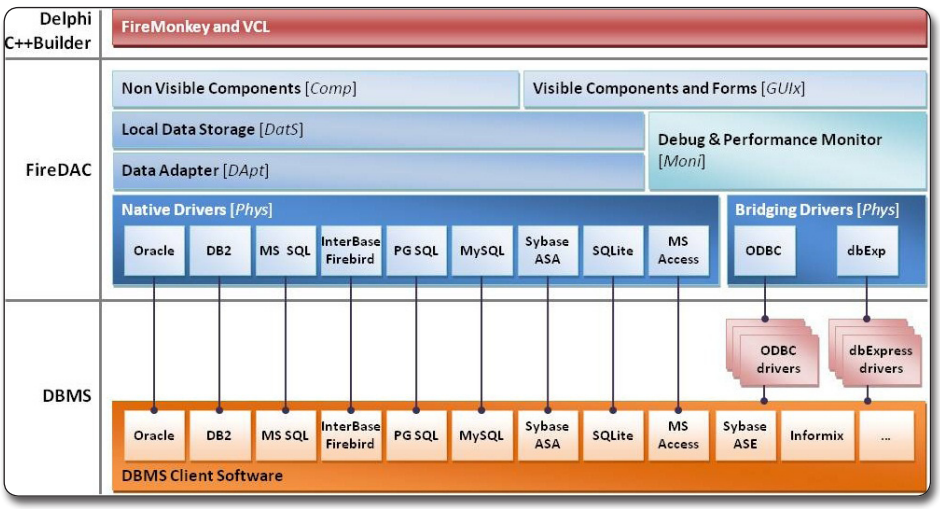

Figura 1. Arquitetura do Framework FireDAC

Nota

O ISQL pode ser acessado no diretório BIN da instalação do Firebird e é uma das alternativas ao IBConsole e IBExpert.

Configurando a Conexão

Começamos criando uma nova aplicação VCL Forms (File>New>VCL Forms Application - Delphi), em seguida, adicionamos um Data Module (File>New>Other..>Delphi Files>Data Module) para trabalharmos com a conexão.

O primeiro passo é a configuração da conexão através do componente TFDConnection que se encontra na categoria FireDAC da Tool Palette. Escolhemos primeiramente o Driver apropriado para o banco de dados utilizado, neste caso o Firebird. Em seguida, definimos os dados da conexão. Na opção Database informamos o caminho do banco de dados físico na máquina ou o Alias. O usuário SYSDBA e a senha masterkey (padrões do Firebird), o protocolo definido é o TCP/IP para acessar o banco se estiver em máquinas separadas. A **Figura 2** ilustra as informações referentes a configuração do componente de conexão.

Se quiséssemos utilizar o mecanismo de Tracing presente no FireDAC, bastaria configurar a opção MonitorBy presente no wizard da **Figura 2** de configuração da conexão. Outra opção seria criarmos um mecanismo de pooling através da propriedade Pooled. Também seria possível optar por capturar a estrutura do banco de dados através dos metadados com a propriedade ExtendedMetadata.

Configuração das Querys

Depois de configurada a conexão, inserimos dois componentes do tipo TFDQuery. Ao adicionarmos o componente, automaticamente sua propriedade Connection recebe a atribuição do componente TFDConnection adicionado anteriormente.

Listagem 1. Script de criação da base de dados de exemplo

```
01. create database 'C:\Exemplo\Database\DBTEST.FDB' page_size 2048
02. user 'SYSDBA' password 'masterkey';
03. create table CLIENTES (ID_CLIENTE int not null, NOME_CLIENTE
    varchar(50) not null, ENDERECO varchar(100), LIMITE_TOTAL
     numeric(10,2), LIMITE_ATUAL numeric(10,2));
04. alter table CLIENTES add constraint PK_CLIENTES primary key (ID_CLIENTE);
05. create sequence GEN_CLIENTES_ID
06. create trigger clientes_bi for clientes
07. active before insert position 0
08. as
09. begin
10.  if (new.id_cliente is null) then
11.    new.id_cliente = gen_id(gen_clientes_id,1);
12. end
```

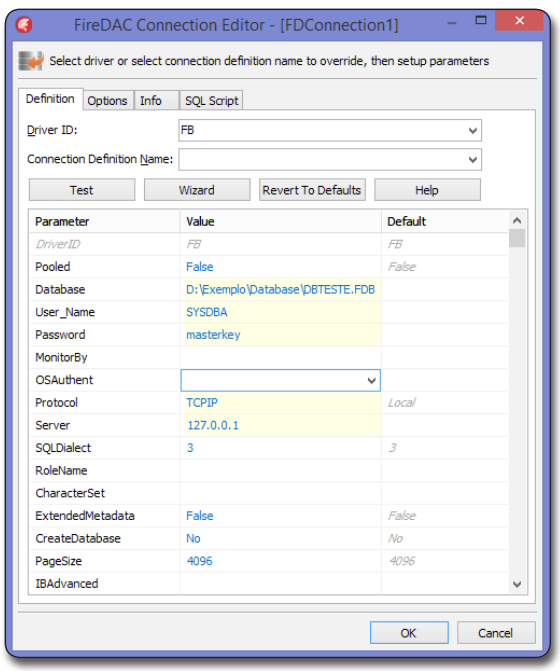

Figura 2. Configuração da conexão através do FDConnection

A ideia é que tenhamos uma Query que fará a consulta de busca dos clientes e outra que será responsável por posicionar um único cliente na tela e fazer as operações de inserção, edição e deleção. Podemos nomeá-los de FDQryLocCliente (consulta) e FDQryCadCliente (para cadastro). Ao darmos duplo clique no TFDQuery, veremos seu editor de Querys que permite o teste ágil de consultas e parâmetros.

A **Listagem 2** ilustra os comandos SQL informados em ambas as Querys. O primeiro é a busca pelo cliente individual por seu identificador e a segunda é adotada para pesquisa do cliente. A função Upper faz com que tanto o campo do banco de dados, quanto o valor do parâmetro sejam convertidos para maiúsculas, facilitando assim a pesquisa.

A primeira alteração necessária é no componente FDQryCadClientes. Para que o FireDAC gere automaticamente os comandos SQL de Insert, Update e Delete, deve-se ativar a propriedade CachedUpdates. Esta faz com que o FDQryCadClientes mantenha um componente TFDMemTable interno para realizar a cache de atualizações, de forma semelhante ao TClientDataSet.

Os comandos de atualização da tabela não serão aplicados automaticamente pelo FireDAC ao chamar os métodos Post/Delete do FDQryCadCliente. De forma análoga ao TClientDataSet, o FDQryCadCliente ficará aguardando uma instrução de ApplyUpdates para iniciar a transação e aplicar a persistência dos dados.

Listagem 2. SQL dos componentes FDQryCadCliente e FDQryLocCliente.

```
1. /* FDQryCadCliente */
2. select *
3. from  CLIENTES
4. where  ID_CLIENTE = :ID_CLIENTE

5. /* FDQryLocCliente */
6. select ID_CLIENTE, NOME_CLIENTE, LIMITE_TOTAL
7. from  CLIENTES
8. where  upper(NOME_CLIENTE) like upper(:NOME_CLIENTE)
```

Nota

A facilidade que temos com o componente TFDQuery ao ativar a propriedade CachedUpdates é muito interessante do ponto de vista prático. No dbExpress, por exemplo, para obter a cache de dados, geração de comandos SQL e acesso a dados, eram necessários três componentes (TSQLDataSet, TDataSetProvider, TClientDataSet). No FireDAC tem-se este trio de componentes unificado em um.

Além disso, pode-se também adicionar os campos TFields ao componente TFDQryCadCliente e assim, configurar individualmente as propriedades da cada um deles, se assim desejado. É possível, por exemplo, nos campos de moeda, configurar a propriedade DisplayFormat com o formato #,##0.00. Assim automaticamente o campo possuirá uma máscara no momento do uso.

Para o componente de pesquisa não é necessário a ativação da propriedade CachedUpdates pois não será realizada nenhum tipo de atualização neste componente, também não é necessária a configuração do TFields.

Configurações adicionais

Há alguns detalhes a configurar antes de finalizar o processo de configuração de acesso a dados. O primeiro detalhe trata-se da necessidade de adicionar uma Unit FireDAC que corresponda ao driver do banco de dados utilizado pela aplicação. O FireDAC nos facilita esse trabalho disponibilizando componentes para cada banco de dados. No nosso caso, devemos adicionar o componente TFDPhysFBDriverLink que se encontra na Tool Palette FireDAC Links. O desenvolvedor pode adicionar a Unit manualmente no Data Module. No exemplo, poderíamos, por exemplo, definir qual a DLL que se comunicará com o banco de dados, através da opção VendorLib. Caso não seja especificada nenhuma, será adotada a dll padrão (fbclient.dll).

O segundo detalhe a ser cuidado é a necessidade de prover um cursor para o banco de dados. Essa é uma exigência do Framework, independente se o banco de dados suporta o uso de cursores de forma implícita ou não. Para isto, basta adicionar

no próprio Data Module o componente FDGUIxWaitCursor, que pode ser encontrado na categoria FireDAC UI da Tool Palette.

Desenvolvimento da interface principal

A criação da interface não será explicada em detalhes por ser simples e estar disponível para download juntamente com o artigo de exemplo. Basicamente foi inserida uma ToolBar com alguns Buttons e atribuído um ImageList para cada botão. Já os campos são DBEdits configurados com seus respectivos TField através de um DataSource vinculado ao nosso FDQryCadCliente. Observe o resultado da interface pronta na **Figura 3**.

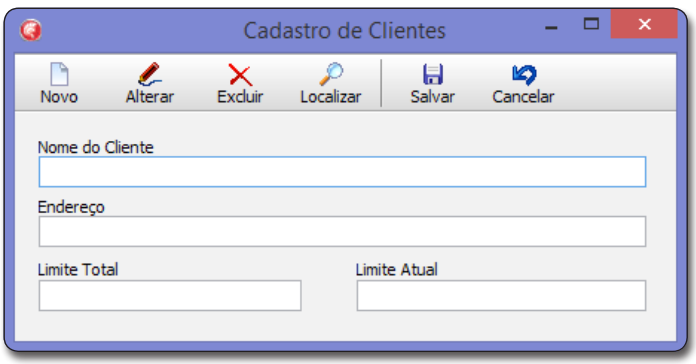

Figura 3. Interface do Cadastro de Clientes

Existem diversas maneiras de controlar a ativação/desativação de controles de tela e barra de ferramentas. Uma das mais simples é utilizando o evento OnStateChange do componente DataSource. Este evento ocorre cada vez que o estado de nosso DataSet, no caso o FDQryCadCliente, sofre alguma alteração. Outra dica com relação aos campos é que podemos atribuir a eles 1 na propriedade Tag para ativarmos/desativarmos de acordo com a necessidade, conforme ilustrado no código da **Listagem 3**.

Pode-se notar que o código apresentado é bem simples. Entre as linhas 05 e 10 realizamos os testes de habilitar / desabilitar os botões através do estado do DataSet em junção com o estado dos botões. A partir da linha 11, percorre-se todos os componentes do formulário verificando a existência da Tag 1. Caso seja encontrada, realiza-se um TypeCast (**BOX 3**) deste componente através da classe TControl, que é justamente a responsável por implementar a propriedade Enabled. Como nota-se, caso esteja em modo de inserção ou edição, habilitamos para o usuário digitar, caso contrário, desativamos. Observe que controla-se os oito componentes da tela em apenas três linhas de código.

As implementações dos eventos de Click dos botões Novo/Alterar/Excluir/Salvar/Cancelar da barra de ferramentas são bastante semelhantes ao que tínhamos anteriormente com o ClientDataSet, deste modo, encontram-se na **Listagem 4**.

Note que essa listagem, embora seja extensa, é relativamente simples. Entre as linhas 01 a 06 (evento OnClick do botão novo), simplesmente é realizada uma inserção através do Append, verifica-se a possibilidade de direcionar o foco para o componente edtNome, fazendo isso caso seja possível. No evento OnClick do botão Alterar, apenas é realizada uma edição (linha 09). Para o botão Excluir, é exibida uma mensagem de confirmação de exclusão do registro (linhas 13 e 14), e caso seja confirmado, o registro é excluído e a persistência aplicada através do ApplyUpdates. No botão salvar é realizada a verificação através da função ConsisteCampos (que será abordada mais adiante na **Listagem 5**). Caso o retorno desta função seja verdadeiro, é realizado o Post e novamente a persistência (linhas 24 e 25). Por último, no evento OnClick do botão Cancelar, apenas é chamado o método Cancel do FDQryCadCliente.

Listagem 3. Ativando e desativando campos e barra de ferramentas

```
01.  procedure TfrmCadCliente.dsClienteStateChange(Sender: TObject);
02.  var
03.    i: Integer;
04.  begin
05.    btnNovo.Enabled := not (dsCliente.State in [dsInsert, dsEdit]);
06.    btnAlterar.Enabled := btnNovo.Enabled and
       i.   (dsCliente.DataSet.RecordCount > 0);
07.    btnExcluir.Enabled := btnAlterar.Enabled;
08.    btnLocalizar.Enabled := btnNovo.Enabled;
09.    btnSalvar.Enabled := dsCliente.State in [dsInsert, dsEdit];
10.    btnCancelar.Enabled := btnSalvar.Enabled;
11.    for i := 0 to ComponentCount -1 do
12.      if Components[i].Tag = 1 then
13.        TControl(Components[i]).Enabled := dsCliente.State in
           i.   [dsInsert, dsEdit];
14.  end;
```

BOX 3. TypeCast

É um recurso no qual consegue-se referenciar um tipo de objeto como se fosse outro. Deste modo, informamos ao compilador que ele deve tratar um objeto como se fosse daquele tipo especificado. Existem duas formas de se fazer isso:

```
TEdit(Sender).text := 'Testando a primeira forma de TypeCast';

(Sender as TEdit).text := 'Testando a segunda forma de TypeCast';
```

A diferença entre as duas é que quando você utiliza-se o primeiro método (TMinhaClasse(Objeto)), o compilador não verifica se sua chamada é válida, ou seja, ele não analisa se objeto pode ser tratado como um da classe TMinhaClasse. É um "hard cast" e somente deve ser usado se você tem certeza absoluta que nunca ocorrerá de ser atribuído um objeto errado, caso contrário, serão gerados erros de Invalid Typecast ou Access Violations. O hard cast (primeira situação), possui melhor performance, pois, não verifica se a chamada é válida, mas, conforme mencionado, deve ser usado com cautela. Além disso, é possível testar o objeto antes do Cast, utilizando o operador "is". Quando usado o tipo "(Objeto as TMinhaClasse)", uma verificação é feita em tempo de execução, checando se a chamada é válida. Caso não seja, novamente uma exceção será gerada. Essa opção é mais segura e mais indicada.

Nota

A propriedade Tag não possui uma funcionalidade predefinida na VCL, então pode ser usada livremente que não fará nenhuma diferença. Ela pode armazenar qualquer valor e serve como um flag para os desenvolvedores.

Conforme mencionado anteriormente, criamos uma função interessante que valida os campos antes de efetivamos a inserção/atualização de registro. O código é exibido na **Listagem 5**.

Listagem 4. Implementação dos eventos OnClick dos botões da barra de ferramentas

```
01.  procedure TfrmCadCliente.btnNovoClick(Sender: TObject);
02.  begin
03.    DM.FDQryCadCliente.Append;
04.    if edtNome.CanFocus then
05.      edtNome.SetFocus;
06.  end;
07.  procedure TfrmCadCliente.btnAlterarClick(Sender: TObject);
08.  begin
09.    DM.FDQryCadCliente.Edit;
10.  end;
11.  procedure TfrmCadCliente.btnExcluirClick(Sender: TObject);
12.  begin
13.    if MessageDlg('Deseja realmente excluir este registro?',
14.      mtConfirmation, [mbYes, mbNo], 0) = mrYes then
15.    begin
16.      DM.FDQryCadCliente.Delete;
17.      DM.FDQryCadCliente.ApplyUpdates();
18.    end;
19.  end;
20.  procedure TfrmCadCliente.btnSalvarClick(Sender: TObject);
21.  begin
22.    if ConsisteCampos then
23.    begin
24.      DM.FDQryCadCliente.Post;
25.      DM.FDQryCadCliente.ApplyUpdates();
26.    end;
27.  end;
28.  procedure TfrmCadCliente.btnCancelarClick(Sender: TObject);
29.  begin
30.    DM.FDQryCadCliente.Cancel;
31.  end;
```

Listagem 5. Validação de campos antes de confirmarmos a inserção/atualização

```
01.  function TfrmCadCliente.ConsisteCampos: Boolean;
02.  var
03.    i: Integer;
04.  begin
05.    Result := True;
06.    for i := 0 to dsCliente.DataSet.FieldCount -1 do
07.    begin
08.      if dsCliente.DataSet.Fields[i].Required then
09.        if (dsCliente.DataSet.Fields[i].IsNull) or
10.          (dsCliente.DataSet.Fields[i].AsString = EmptyStr) then
11.        begin
12.          MessageDlg(Format('Campo %s é obrigatório e não foi informado',
13.          [dsCliente.DataSet.Fields[i].DisplayLabel]),mtWarning,[mbOK],0);
14.          Result := False;
15.          Break;
16.        end;
17.    end;
18.  end;
```

Nesse código temos um laço for (linha 06) que percorre os campos TField do DataSet a ser persistido, validando se aquele determinado campo é requerido pelo banco de dados (linha 08). Caso seja obrigatório, é validado se este é ou não nulo (linha 09), ou se no caso de uma string, se seu conteúdo é vazio (linha 10). Se uma das condições for verdadeira, mostra uma mensagem para o usuário informando que o campo é obrigatório (linhas 12 e 13).

Essa abordagem de verificação direta na aplicação cliente é interessante porque evita que recebamos uma mensagem de erro do banco de dados, e por ser uma validação local, em memória, é muito mais rápida.

A primeira otimização do formulário a ser realizada é mostrada no código da **Listagem 6.** Assim sendo, é realizada a verificação do estado do DataSet (linha 03) ao usuário tentar fechar a tela principal (evento OnClose). Caso o DataSet esteja em edição ou inserção, solicita-se a confirmação de saída (linha 05), fechando o DataSet caso confirmada (linha 06). Com isso, evitamos vazamentos de memória.

Listagem 6. Verificação de saída da aplicação

```
01.  procedure TfrmCadCliente.FormClose(Sender: TObject;
02.   var Action: TCloseAction);
03.  begin
04.    if dsCliente.State in [dsInsert, dsEdit] then
05.    begin
06.      if MessageDlg('Deseja sair sem salvar as alterações?', mtConfirmation,
         [mbYes, mbNo], 0) = mrYes then
07.        DM.FDQryCadCliente.Cancel
08.      else Action := caNone;
09.    end;
10.  end;
```

Criando o formulário de pesquisa

Para a pesquisa do cliente criaremos outro formulário. Sendo assim, acione o menu "File>New>VCL Form – Delphi" e adicione componentes, configurando o Layout conforme a **Figura 4**. Nele foram adicionados um Edit para pesquisa, um grid para resultados, além de uma StatusBar que nos indica o número de registros encontrados na pesquisa. Também foi adicionada uma propriedade chamada IdSelecionado que será utilizada para escrever o identificador do cliente selecionado no grid, para posterior utilização da tela principal.

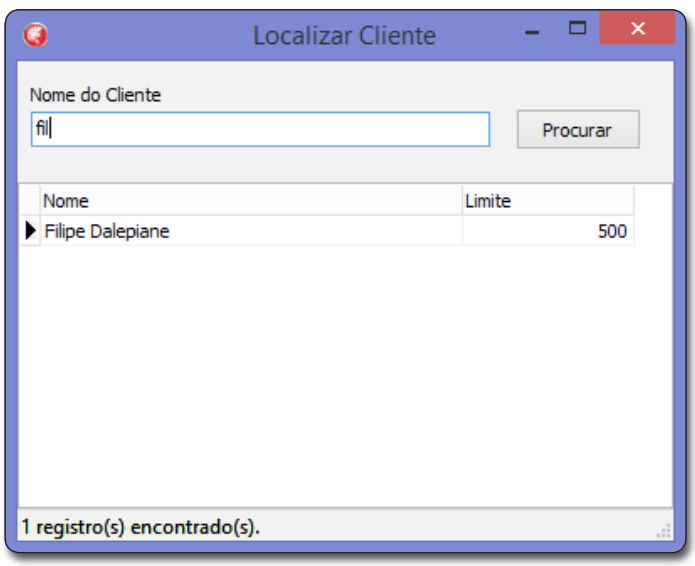

Figura 4. Formulário de pesquisa de clientes

Pode-se realizar a ordenação dos registros do grid quando o usuário pressionar o título do campo. Isto é mostrado no código da **Listagem 7**. Esta ordenação faz uso da sintaxe semelhante ao que tínhamos com o ClientDataSet e é realizada de forma muito rápida, afinal os dados estão em cache.

Uma boa prática em campos de pesquisa onde pode-se haver milhares de registros é a obrigatoriedade de um número de caracteres mínimo de pesquisa, evitando travamentos e lentidão da aplicação. Neste exemplo isso foi codificado no evento OnKeyUp do Edit de pesquisa. Este evento ocorre logo após o usuário digitar o caractere, validando posteriormente o tamanho do texto digitado e de acordo com ele habilitamos ou não o botão procurar. Note que além do evento OnKeyUp, é exibido o evento OnClick do botão procurar no código da **Listagem 8**.

Listagem 7. Ordenação dos registros em memória

```
01.  procedure TfrmLocCliente.grdPesquisaTitleClick(Column: TColumn);
02.  begin
03.    DM.FDQryLocCliente.IndexFieldNames := Column.FieldName;
04.  end;
```

Listagem 8. Verificação para habilitar pesquisa e evento do botão pesquisar

```
01.  procedure TfrmLocCliente.edtPesquisaKeyUp(Sender: TObject; var Key: Word;
02.    Shift: TShiftState);
03.  begin
04.    btnProcurar.Enabled := Length(Trim(edtPesquisa.Text)) >= 3;
05.  end;
06.  procedure TfrmLocCliente.btnProcurarClick(Sender: TObject);
07.  begin
08.    DM.FDQryLocCliente.Close;
09.    DM.FDQryLocCliente.Params[0].AsString := '%' + edtPesquisa.Text + '%';
10.    DM.FDQryLocCliente.Open();
11.    if DM.FDQryLocCliente.RecordCount > 0 then
12.      StatusBar1.SimpleText := DM.FDQryLocCliente.RecordCount.ToString() +
13.        ' registro(s) encontrado(s).'
14.    else StatusBar1.SimpleText := 'Nenhum registro encontrado';
15.  end;
```

No código apresentado podemos notar o evento OnClick do botão de procura. Observe que o procedimento é bem simples. Entre as linhas 08 a 10 o DataSet é aberto, passando como parâmetro o conteúdo do campo de pesquisa concatenado com o '%'. Na linha 11 é verificado se a busca retornou registros, exibindo assim a quantidade de registros na StatusBar, caso contrário, é mostrada a ausência de registros. Uma observação interessante é que na linha 12 temos a utilização de um recurso relativamente novo do Delphi, chamado de Class Helper (**BOX 4**). Ao invés de adotar a função IntToStr para concatenar o total de registros com o texto, usa-se a Class Helper TIntegerHelper.ToString, que realiza automaticamente a conversão, tornando bem mais simples a tarefa, além de tornar a sintaxe mais agradável.

Ao acionar o duplo clique do grid, apenas é atribuído o ID_CLIENTE selecionado para a propriedade IdSelecionado da classe de formulário. Agora, basta codificar a chamada da tela de pesquisa na janela principal, conforme mostra a **Listagem 9**.

BOX 4. Class Helper

Class Helper é um tipo que, quando associada à outra classe, introduz métodos adicionais e propriedades que podem ser utilizadas em um contexto da classe associada (ou seus descendentes). Essa característica da linguagem e do compilador Delphi traz a grande vantagem de permitir estender novas funcionalidades em classes já existentes, podendo assim adicionar novos métodos e propriedades sem a necessidade de se criar uma nova classe.

Listagem 9. Chamada do formulário de pesquisa e posicionamento do registro

```
01.  procedure TfrmCadCliente.btnLocalizarClick(Sender: TObject);
02.  begin
03.    Application.CreateForm(TfrmLocCliente, frmLocCliente);
04.    try
05.      frmLocCliente.ShowModal;
06.      if frmLocCliente.IdSelecionado > 0 then
07.      begin
08.        DM.FDQryCadCliente.Close;
09.        DM.FDQryCadCliente.ParamByName('ID_CLIENTE').AsInteger :=
10.          frmLocCliente.IdSelecionado;
11.        DM.FDQryCadCliente.Open();
12.      end;
13.    finally
14.      frmLocCliente.Release;
15.    end;
16.  end;
```

No código apresentado observamos o objeto Application, que é uma variável global na unit Vcl.Forms (linha 03), criando assim o formulário através do método CreateForm. Observe que é validado se a propriedade IdSelecionado é maior que 0 (linha 06), fazendo assim com que o a Query de cadastro abra o registro selecionado (linhas 08 a 11). A seguir, na linha 14, o formulário é destruído.

Nota

O método Release é utilizado quando trabalha-se com Forms pois, este pertence à classe TCustomForm. Sendo assim, o Release deve ser usado quando queremos liberar um Form dentro de outro. A diferença do método Release para os métodos Free ou FreeAndNil é que ele executa as mensagens e eventos que estiverem em fila daquele formulário antes de ser destruído, e, só depois dispara internamente o método Free.

Configurando o Data Module para FireMonkey

Como sabe-se, a VCL e o Firemonkey são plataformas gráficas incompatíveis, ou seja, componentes criados para VCL não podem ser adicionados em formulários FireMonkey e vice-versa. Mas existe uma maneira de utilizarmos o mesmo Data Module para as duas plataformas. Para isso, devemos fazer algumas pequenas adaptações.

A propriedade que nos permite que possamos utilizar o Data Module em ambas arquiteturas é a propriedade ClassGroup. Nela são informados quais os tipos de componentes que poderão ser adicionados no Data Module. Tem-se então três opções:

- System.Classes.TPersistent: permite somente os componentes padrão das duas plataformas.
- FMX.Controls.TControl: além dos componentes padrão das duas plataformas, permite componentes não visuais da plataforma FMX.

• VCL.Controls.TControl: além dos componentes padrão das duas plataformas, permite componentes não visuais da plataforma VCL.

Note que ao trocar as opções das propriedades, os componentes disponíveis na Tool Palette vão se alterando para respeitar a compatibilidade. Desta maneira, para não termos nenhum erro na aplicação, deve-se utilizar a opção System.Classes.TPersistent, assim, o Data Module será compatível tanto com a VCL quanto o FireMonkey. Ficarão disponíveis apenas os componentes que são independentes de plataforma.

Criando um projeto FireMonkey

Criamos um novo projeto do tipo FireMonkey HD (File> New>Firemonkey Desktop Application). Nele adicionamos ao projeto o mesmo Data Module criado anteriormente. A barra de ferramentas conterá os mesmos botões da VCL e a mesma codificação, por isso, não será exibida. O restante da codificação será praticamente a mesma, mudando apenas a ativação/desativação de controles, os componentes visuais e a maneira de realizar a ligação com o banco de dados.

Aqui vale lembrar que em projetos FireMonkey não há controles DataWare (TDBEdit, TDBGrid e etc.) para efetuar a ligação dos controles de tela aos TFields dos DataSets. Ao invés deles, é adotado o mecanismo de LiveBindings (**BOX 5**) criado especialmente para esta finalidade. Desta maneira, no lugar de um componente TDBEdit, teremos então um TEdit convencional.

BOX 5. LiveBindings

O LiveBinding é um mecanismo de DataBinding que foi incorporado à versão XE2 do Delphi e vem evoluindo com o passar das versões. Tem por finalidade realizar a ligação de fontes distintas de dados através de expressões. Estas expressões são relacionais, conhecidas como Binding Expressions, podendo ser unidirecionais ou bidirecionais, definindo o sentido do fluxo de dados na ligação.

- Unidirecional: a alteração é feita apenas na origem, ou seja, a propriedade de destino apenas reflete os dados da origem.
- Bidirecional: a alteração é feita na origem e no destino, ou seja, qualquer alteração é refletida em ambos.

Por meio destas expressões torna-se possível ligar qualquer objeto, simplesmente definindo a expressão de ligação entre eles.

Para realizar a adição dos controles de tela, pode-se utilizar o recurso do LiveBinding chamado de LiveBindings Wizard. Para acessá-lo, basta acionar o menu View>LiveBindings Designer. Nesse momento escolhemos o último botão da barra de ferramentas da esquerda da janela. O processo aqui é bem simples, basta seguir o passo a passo, escolhendo primeiramente a opção "Link a control with a field". Em seguida, escolhemos o componente na qual desejamos fazer a conexão (TEdit neste caso). O próximo passo é selecionar o DataSet (FDQryCadCliente) que será vinculado ao componente. Por último, basta informar o campo desejado. Podemos ainda solicitar a adição de um Label para descrever o nome ao campo.

Basta seguir os mesmos passos para os demais campos do formulário. Após o processo, teremos a mesma estrutura que a aplicação criada na VCL. A **Figura 5** ilustra o processo de Binding.

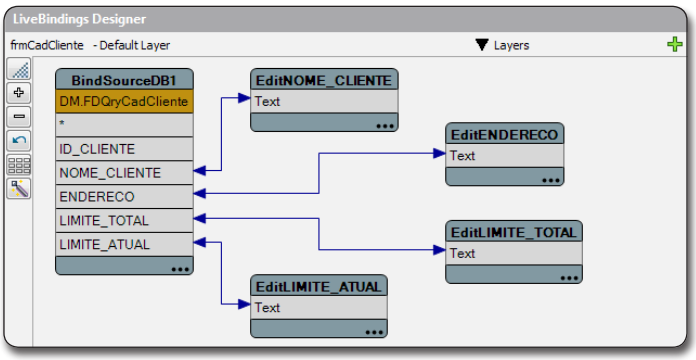

Figura 5. Binding entre o DataSet e os componentes de tela

Já na visualização do formulário, podemos observar dois novos componentes criados automaticamente pelo Delphi, são eles o TBindSourceDB e o TBindingsList. O TBindSourceDB é muito semelhante ao DataSource presente na VCL, inclusive possui internamente um DataSource e um DataSet para realizar esta comunicação. Pode-se então codificar a ativação/desativação dos controles de tela e barra de ferramentas através do evento OnStateChange, como anteriormente realizado com o DataSource. Já o componente TBindingsList é o responsável por fazer a ligação entre dois objetos, TField e TEdit. É nele que fica a expressão de ligação, o DataBinding.

Ao entrar no editor de ligações do TBindingsList, é possível visualizar a maneira com que o Delphi realiza a ligação entre as duas fontes de dados (DataSet e Edit). Para cada campo foi criado um objeto do tipo TLinkControlToField, conforme mostra a **Figura 6**. Neste link pode-se fazer algumas otimizações, como a formatação do campo moeda, na propriedade CustomFormat.

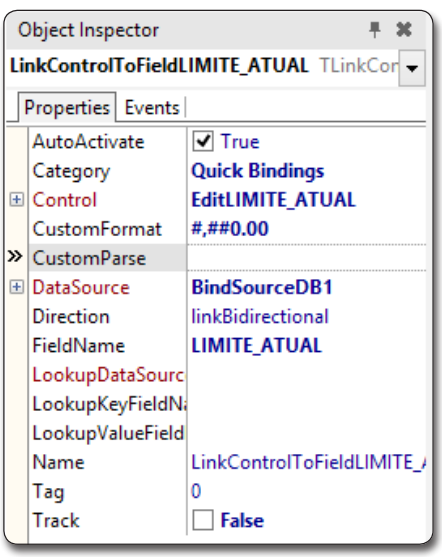

Figura 6. Link criado entre o componente Edit e o Field do DataSet

Para a criação do formulário de pesquisa no FireMonkey, é utilizado novamente o recurso de LiveBindings para efetuar o carregamento dos dados no grid. Sendo assim, abra novamente o LiveBindings Designer, mas desta vez, selecione a opção "Link a grid with a data source". Após escolhermos o tipo de grade para exibição, que neste caso foi o TStringGrid. Por último, é escolhida a fonte de dados (FDQryLocCliente).

Como ocorreu na tela principal temos a criação dos componentes TBindSourceDB e TBindingsList. Mas desta vez, se repararmos no item criado no BindingsList, veremos que o tipo de objeto de ligação criado é TLinkGridToDataSource, onde fica especificado o a ligação entre os componentes StringGrid e DataSource, conforme mostra a **Figura 7**.

Nota

Um objeto da classe TStringGrid representa um controle de grid que pode ser usado em suas aplicações Delphi para simplificar o processo de manipular strings e objetos associados a esta. O uso mais frequente de um controle TStringGrid é quando se deseja apresentar um conteúdo texto em um formato tabular. Este controle fornece muitas propriedades de customização da aparência, assim como eventos e métodos que usufruem o máximo da organização tabular da grid ao responder às ações do usuário.

Nota

O formulário de pesquisa de clientes no FireMonkey não será apresentado no artigo mas pode ser visto ao realizar o download do código fonte.

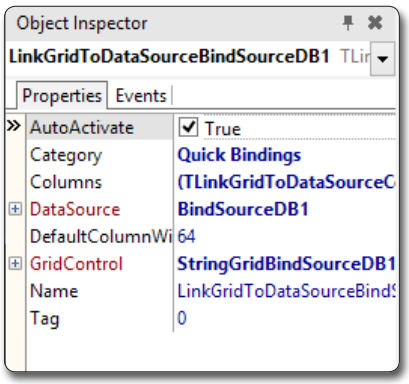

Figura 7. Link entre o componente StringGrid e a fonte de dados

Convertendo o projeto para Metropolis

Todo e qualquer formulário da VCL e FireMonkey possui um mecanismo rápido e simples de convertermos interfaces para o estilo Metropolis UI (ver **BOX 6**). Podemos a qualquer momento clicar com o botão direito no formulário, acionando a opção "Convert to Metropolis UI".

Neste momento temos uma mudança instantânea na interface, o Delphi faz isso através do componente TStyleBook, que carrega o estilo de interface do Metropolis. Esses estilos podem ser acessados no diretório de arquivos extras da Embarcadero que geralmente se encontram em "C:\Users\Public\Documents\Embarcadero\Studio\14.0\Styles". Observe o formulário convertido na **Figura 8**.

Figura 8. Interface Firemonkey (Metropolis UI)

BOX 6. Metropolis UI

Metro UI, ou simplesmente Metro, foi o codinome dado à nova interface de usuário desenvolvida pela Microsoft para o sistema operacional Windows 8. Dentre suas características, ela remete a termos como simplicidade, clareza e modernidade. Este tipo de interface foi primeiramente visto na plataforma móvel Windows Phone 7, através do uso de uma linguagem de design toda baseada em tipografia. Em agosto de 2012 o termo "Metro" foi extinto pela Microsoft, sob a justificativa de que se tratou apenas de uma denominação inicial. Todavia, ainda é notório o vínculo existente, no meio popular, entre o referido termo e o sistema operacional.

A produtividade encontrada com o trabalho com banco de dados no Delphi, agora com FireDAC é incomparável a qualquer outra ferramenta de desenvolvimento. O FireDAC contribuiu ainda para que o Delphi se tornasse ainda mais RAD.

O dbExpress também será suportado pelo Delphi, até porque é a tecnologia que junto com os componentes Indy, formam o DataSnap. No entanto, é possível utilizar o DataSnap juntamente com o FireDAC.

Muitas empresas ficam inseguras quando é lançada uma nova tecnologia, porém, o FireDAC já existe há muito tempo, antes chamado de UniDAC. Para os desenvolvedores que o utilizavam, ficará ainda mais fácil a migração, pois, a Embarcadero fez poucas mudanças até o momento.

Autor

Filipe Dalepiane
filipe.dalepiane@gmail.com
http://filipedalepiane.blogspot.com.br/
Analista Desenvolvedor de Sistemas numa Universidade.Colunista das revistas Clube Delphi e .NET Magazine.Desenvolvedor Delphi e C#, utilizando bancos de dados SQLServer e Firebird. Bacharel em Ciência da Computação.

Você gostou deste artigo?

Dê seu voto em ***www.devmedia.com.br/clubedelphi/feedback***

Ajude-nos a manter a qualidade da revista!

Somos tão apaixonados por tecnologia que o nome da empresa

diz tudo.

Porta 80 é o melhor que a Internet pode oferecer para sua empresa.

Já completamos 8 anos e estamos a caminho dos 80, junto com nossos clientes.

Adoramos tecnologia.
Somos uma equipe composta de gente que entende e gosta do que faz,

assim como você.

Estrutura

100% NACIONAL. Servidores de primeira linha, links de alta capacidade.

Suporte diferenciado

Treinamos nossa equipe para fazer mais e melhor. Muito além do esperado.

Serviços

Oferecemos a tecnologia mais moderna, serviços diferenciados e antenados com as suas necessidades.

1-to-1

Conhecemos nossos clientes. Atendemos cada necessidade de forma única.
Conheça!

Porta 80
WEB HOSTING

Hospedagem | Cloud Computing | Dedicados | VoIP | Ecommerce | Aplicações | Streaming | Email corporativo

porta80.com.br | comercial@porta80.com.br | twitter.com/porta80

SP 4063-8616 | RJ 4063-5092 | MG 4063-8120 | DF 4063-7486

Ele está trabalhando em seu saque e não na sua nova aplicação.
Mais de 95%* dos times de desenvolvimento e testes de aplicativos relatam tempos de espera e atrasos para acessar os sistemas que necessitam para realizar suas atividades. A Virtualização de Serviços elimina estas dependências gerando ambientes simulados similares à realidade, permitindo o desenvolvimento em paralelo. Isto significa que suas aplicações serão lançadas mais rapidamente, com maior qualidade e menor custo. Game over!
Conheça mais em ca.com/br/GoDevOps
*De acordo com o estudo 2012 North America/Europe Service Virtualization Study, Coleman-Parkes.
ca technologies